①

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE STABLE AND PRECISE MOTION CONTROL FOR AN AUTONOMOUS MOBILE ROBOT

by

Ten-Min Lee

March 1994

Thesis Advisor: Yutaka Kanayama

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE March 1994	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE
THE STABLE AND PRECISE MOTION CONTROL FOR AN AUTONOMOUS MOBILE ROBOT

5. FUNDING NUMBERS

6. AUTHOR(S)
Lee, Ten-Min

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The major problem addressed by this research is how to develop a motion control algorithm for stable and precise control of the motion of an autonomous mobile robot.

The approach taken was to clearly define the robot's motion descriptions and to design a high-level, machine independent robot control language called MML (Mode-based Mobile robot Language).

The results are that the robot can implement line to line, line to circle, circle to circle path tracking or the combinations of these. Based on the motion control algorithm which was developed in this thesis, the robot is able to use external sensors to execute complicated missions such as obstacle avoidance (sonar is used in this thesis work).

14. SUBJECT TERMS
Motion Control, Path-Tracking

15. NUMBER OF PAGES
94

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THE STABLE AND PRECISE MOTION CONTROL FOR AN AUTONOMOUS MOBILE ROBOT

by

Ten-Min Lee
Lieutenant Colonel, R.O.C. Army

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

March 1994

Author: Lee Ten-Min

Ten-Min Lee

Approved By: Yutaka Kanayama

Yutaka Kanayama, Thesis Advisor

[illegible]

Yuh-jeng Lee, Second Reader

Ted Lewis

Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The major problem addressed by this research is how to develop a motion control algorithm for stable and precise control of the motion of an autonomous mobile robot.

The approach taken was to clearly define the robot's motion descriptions and to design a high-level, machine independent robot control language called MML (Mode-based Mobile robot Language).

The results are that the robot can implement line to line, line to circle, circle to circle path tracking or the combinations of these. Based on the motion control algorithm which was developed in this thesis, the robot is able to use external sensors to execute complicated missions such as obstacle avoidance (sonar is used in this thesis work).

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

There are many different applications for the autonomous mobile robot. Especially in military operations, a robot can execute many dangerous missions which human beings may possibly risk their lives. Removing mines, taking photographs in hazardous areas or even fighting with detected targets are just a few of the possible applications.

The motion control theory is the basis of successful control the robot (autonomous wheel controlled vehicle). Based on the robot's smooth motion control algorithm which has been developed in this thesis work, we can precisely and stably control the robots' motion. After the success in controlling the robot's motion, the robot will be able to execute some complicated motion missions such as path tracking and obstacle avoidance. To allow users to easily use the robot ('Yamabico-11' is the experimental robot for this thesis work), the robot control language MML (model-based mobile robot language) was investigated and improved in this thesis work.

The path tracking mission includes line to line tracking, line to circle tracking, line to parabola tracking, circle to circle tracking or all the above motion combinations. A typical path tracking mission can be described as this: The robot is setting by the road and moves to the center of the road and keeps moving for two blocks, then it makes a left turn to another road and moves for one block later and parks on the side of the road (Figure 1).

The obstacle avoidance mission will use the path tracking algorithm and external sensors to allow the robot to avoid possible obstacles (which may exist in its original path) and return to its original path after it safely passes obstacles (Figure 2).

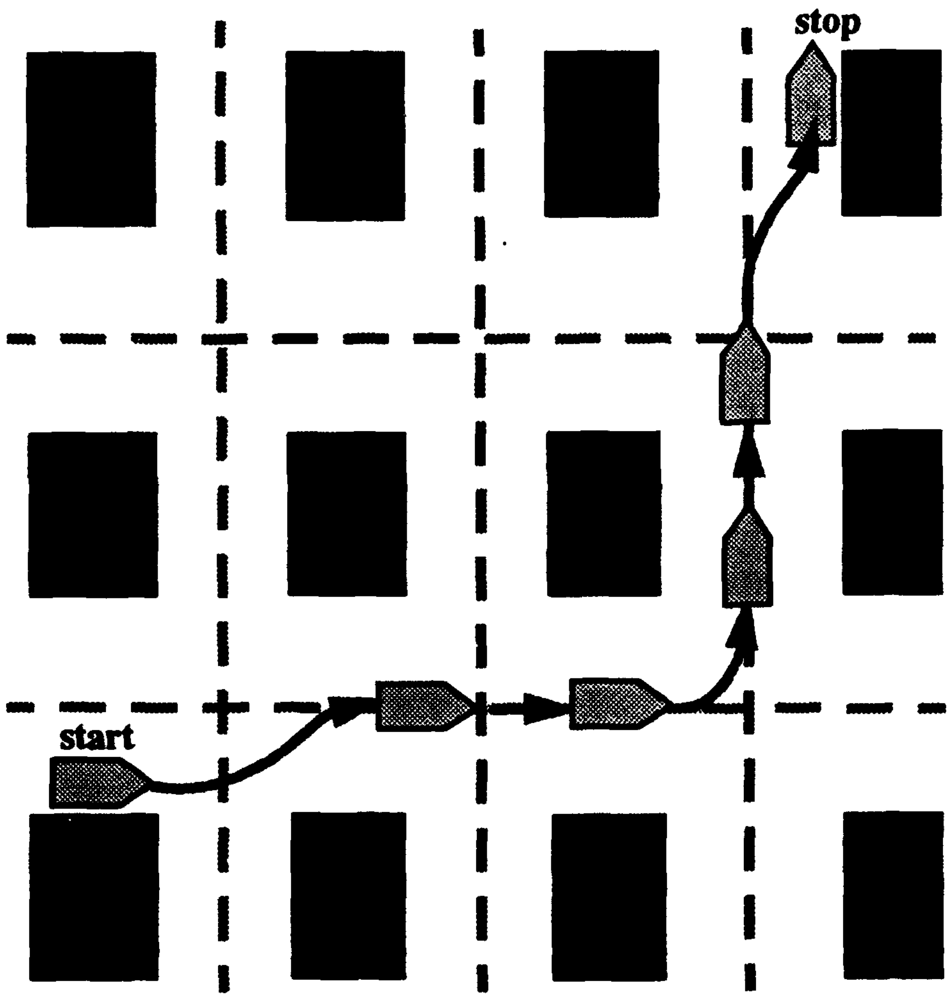


Figure 1: A path tracking mission.

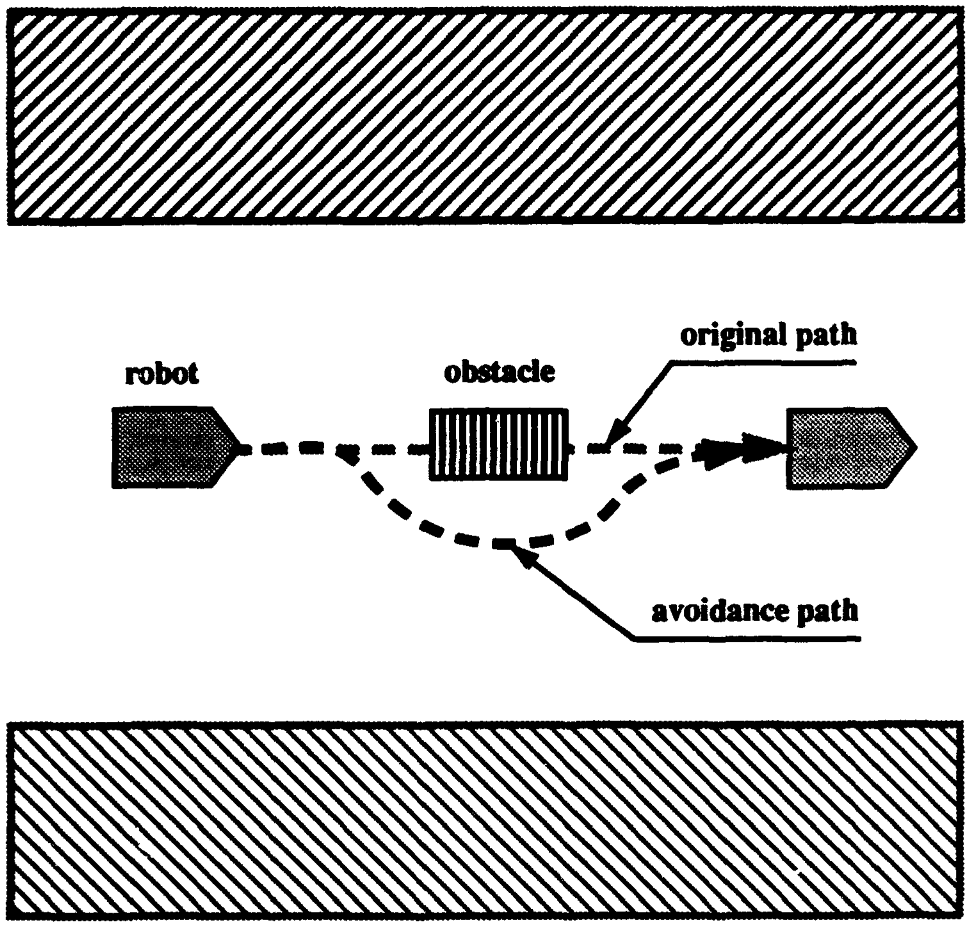


Figure 2: An obstacle avoidance mission.

II. PROBLEM STATEMENTS

The problem investigated in this thesis is how to efficiently describe the robot's motions and how to precisely and stably control the motions of the autonomous mobile robot "Yamabico-11". This problem is subdivided into the following subproblems:

1. Is there a unified and user-friendly standard high-level language to describe robot's motions?

2. How to clearly and efficiently describe the robot's motions (motion description)?

3. How to let the robot know its current position (odometry function)?

4. How to compute the robot's next translational and rotational speed for the purpose of tracking a given path?

5. How to precisely control motors to let the robot reach the desired speed?

III. MML REAL-TIME CONTROL SOFTWARE ARCHITECTURE

The control language used for programing Yamabico-11 is MML (model-based mobile robot language). MML library functions are categorized into motion, sonar, I/O, geometry,.... For motion control, we need short sampling time (in Yamabico-11's case, the robot takes 10 *ms* for the motion feedback control). Since motion control takes so much of the CPU time, we separate this low-level control process from robot's main computation.

In Figure 3, the MML real-time control software architecture is divided into two different process levels, one is the foreground process for the user program and the other is the background process for the motion feed-back control functions.

The user interface functions include two types of motion functions, immediate and sequential functions. Users can use these functions to control the robot without knowing the low level functionality. These user interface functions are the tools which will be provided to users to allow them to command the robot. In our autonomous mobile robot path-tracking mission which was described in Chapter I, the user only needed two kinds of motion functions such as *line* and *bline* functions to let the robot accomplish its mission. The fewer commands needed, the easier to write the user program for the user. The user interface functions and the robot's motion description will be discussed in detail in Chapter IV.

There is an instruction buffer between the foreground process and the background process. It is a circular queue design. The user interface functions used in the user program will be put into the queue. The background process functions will be executed every 0.01 seconds (10 *ms*) once the mission starts. The motion control program in the background process will get the instructions from the queue whenever it is needed. The details of the motion control program in background will be described in Chapter V.

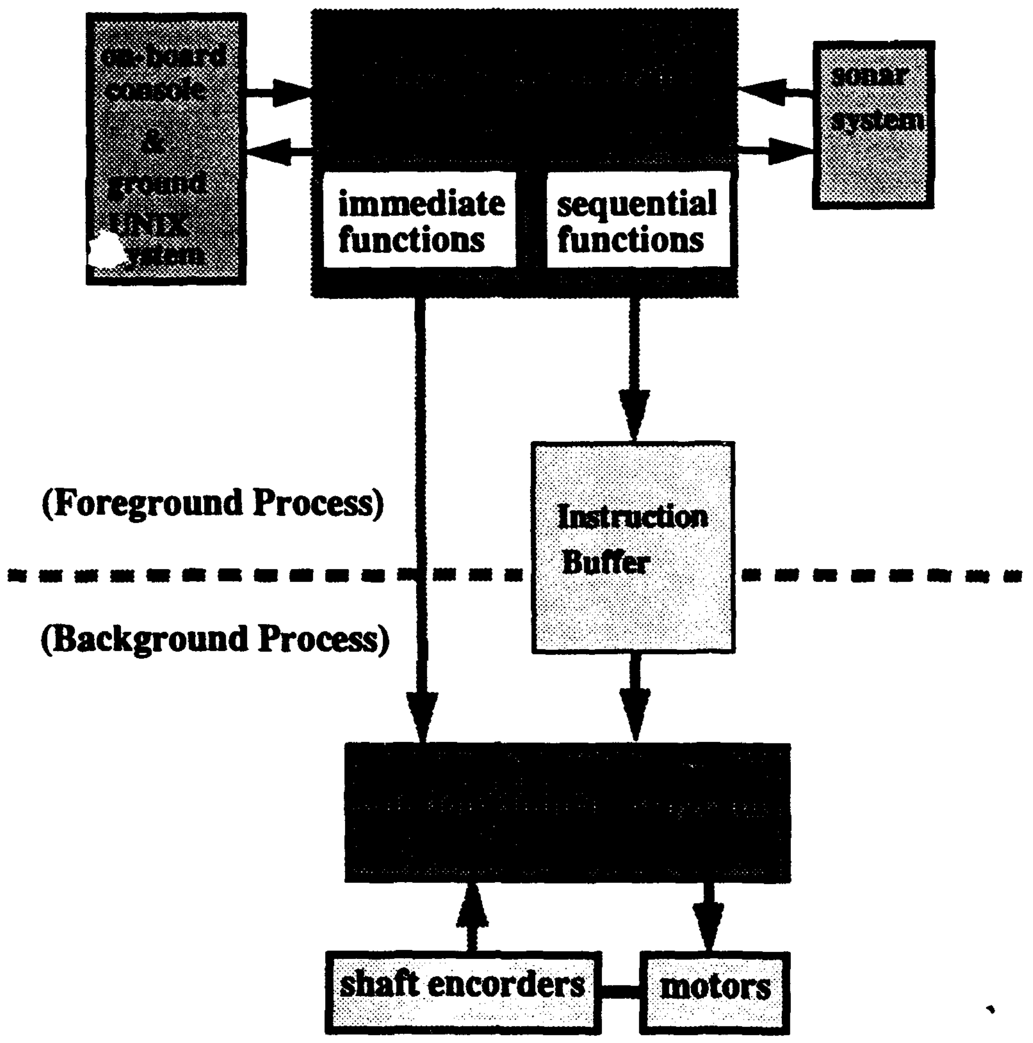


Figure 3: MML Real-Time Control Software Architecture.

IV. LOCOMOTION FUNCTIONS IN THE MML LANGUAGE

A. MOTION DESCRIPTION BY PATH ELEMENTS

In order to clearly describe the robot's motion, some definitions and conventional terms need to be introduced.

Define the robot's path by the geometrical method. A configuration q stands for a triple

$$q = (p, \theta, \kappa),$$

where p is a point, θ is an orientation, and κ is a curvature. A configuration q is a representation of a directed line or a directed circle, when κ is less than zero it is a clockwise circle and it is a counterclockwise circle when κ is greater than zero, (Figure 4). A directed parabola is represented by a directed line q (directrix) and a point p (focus), (Figure 5). A directed cubic spiral path element is defined by two configurations q_1 and q_2 (in this case its curvature κ is not important), (Figure 6).

A path is a sequence $(e_1, e_2, e_3, \ldots, e_n)$, where each e_i is an element. For an arbitrary configuration q and a point p, either one of the following is said to be an element (e):

configuration, parabola, cubic_spiral.

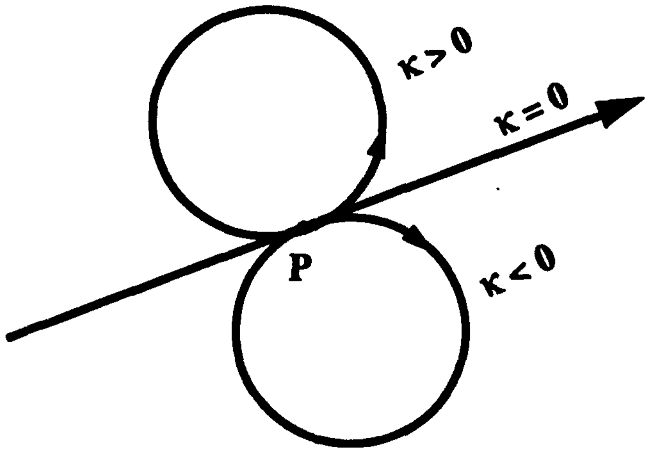


Figure 4: A configuration represents a line or a circle (directed).

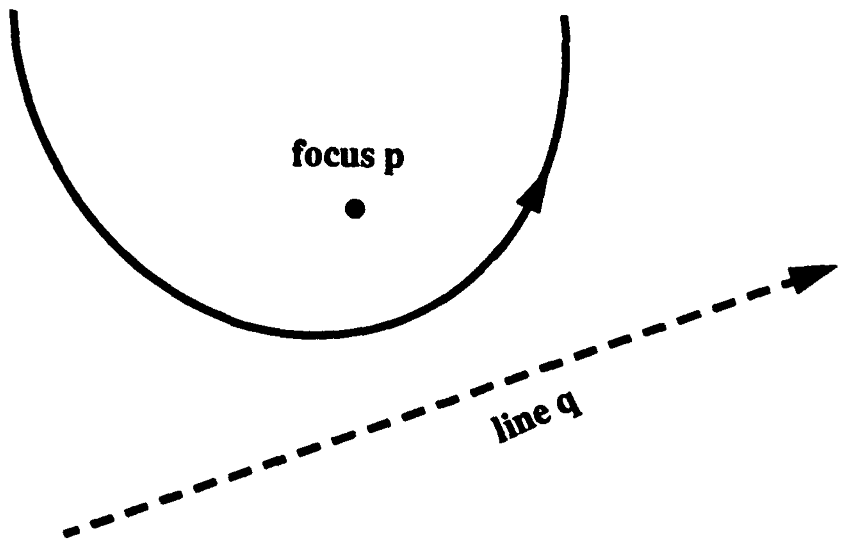


Figure 5: parabola.

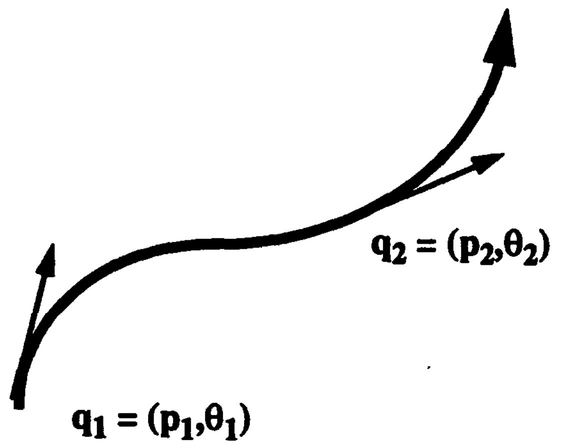


Figure 6: cubic_spiral.

B. USER INTERFACE FUNCTIONS

The meaning of each element *e* is defined as follows (each element means a directed simple path if *e* is not a *configuration*).

1. *line(q)*. This path segment does not have any endpoints.

2. *Parabola(q)* means a directed parabola determined by the focus *p* and the directrix *q*. (The curvature part of κ of *q* is ignored.)

3. *forward_line(q)* means a part of element *line(q)*. It has a start *p*.

4. *backward_line(q)* means a part of element *line(q)*. It has an end *p*.

5. *config(q)* does not mean a directed path segment by it self. It must have elements which specifies another configuration in the previous and the following position in its path. A pair of *config(q)* define a cubic_spiral path segment.

C. TRANSITIONS BETWEEN PATH ELEMENTS

There are legitimate transitions between path elements. If two adjacent elements specified by two sequential functions are intersecting, the robot leaves a "leaving point" on the first element and, after then, tracks the second one. If there is no intersections between two adjacent path element, the robot immediately leaves the first one.

The "leaving point" is the last point on the first element which does not make the following trajectory oscillatory. The tracking algorithm realizes the critical damping solution in all cases. This leaving point calculation is a time consuming task and is being done in real-time in the MML software system. Table1 shows the permissible types of motion function transitions.

TABLE 1: PERMISSIBLE TRANSITIONS IN MOTION FUNCTIONS

From / To	line	parabola	backward_line	Forward_line	Config
Line	TR	TR	TR	--	--
Parabola	TR	--	TR	--	--
Forward_line	TR	TR	TR	--	--
Backward_line	TR E	TRE	TRE	CS	CS
Config	--	--	--	CS	CS

* TR: normal transition. TRE: transition at the endpoint.
CS: cubic spiral. --: not permissible.

D. TRANSITIONS BY IMMEDIATE MOTION FUNCTIONS

There are immediate motion functions which affect the robot's motion immediately without being stored in the instruction buffer.

"*stop0*", "*skip*", "*halt*", "*speed0*" and "*set_c*" are examples of immediate functions. when "*stop0*" and "*skip*" are used, the robot leaves the current sequential motion function to execute this new immediate function.

E. MOTION FUNCTIONS IN YAMABICO-11

1. Line

Syntax: void line(q)

Configuration q;

Description:

The argument q = (p, θ, κ) specifies a straight line or a circular arc. Ba
the robot is supposed to follow this directed path element. The robot leaves this element when it comes to a leaving point or when an immediate motion function are called. Figure 7 shows the robot tracks a *line* path element from its previous configuration. The robot's speed is automatically reduced to allow the robot to make sharp turns. This is reflected by the dependency between κ and the robot's speed. In simple terms, the robot's speed must be reduced to allow it to move safely with larger values of κ. When next path element is given, the robot leaves the current path element in the manner described in section C and D.

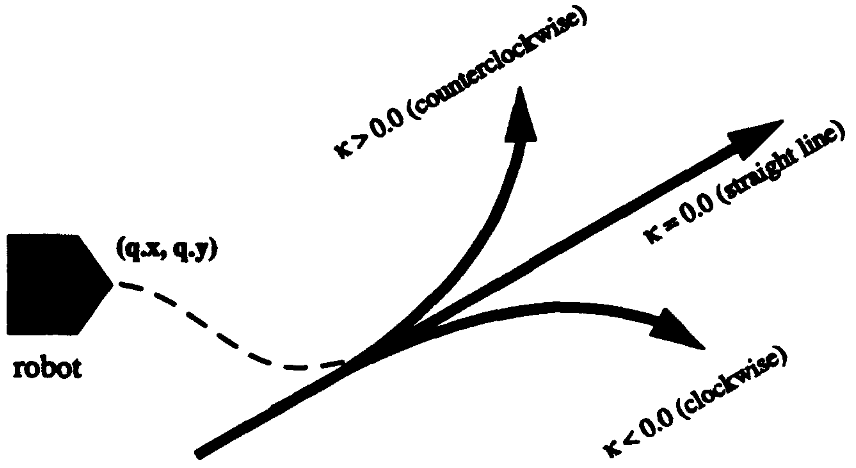


Figure 7: The line function.

2. Fline (forward line)

Syntax: void fline(q)

Configuration q;

Description:

The argument q = (p, θ, κ) specifies a straight line or a circular arc. Basically the robot is supposed to follow this directed path element. This function makes the robot track this path element from the point P, i.e., the robot passes through the configuration (Figure 8). To properly use this function, the robot's last path element needs a specified end-configuration. Any path segment which does not provide a specified end-configuration will not allow to combine with *fline* function.

If a *fline* function came after a *line* function, it is an error usage of *fline* function. Because a *line* function does not have a end-configuration.

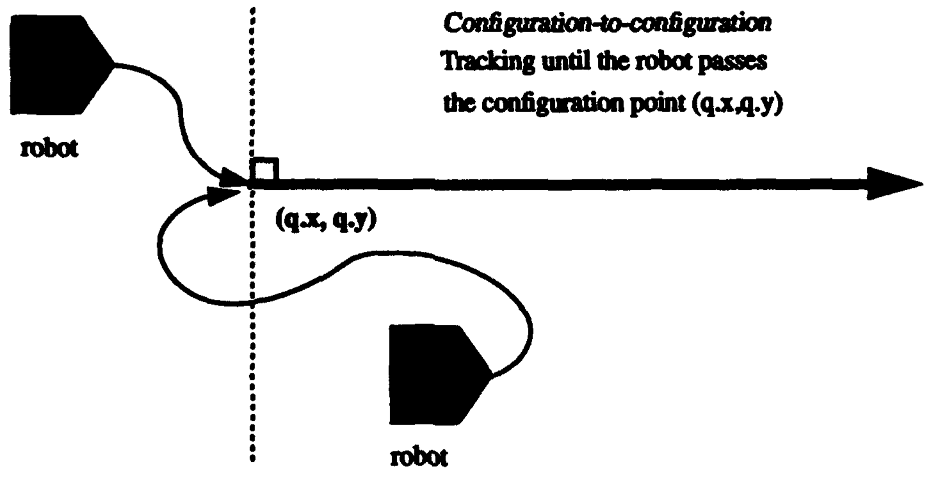

Configuration-to-configuration
Tracking until the robot passes
the configuration point (q.x,q.y)
robot
(q.x, q.y)
robot

Figure 8: The fline function.

3. Bline (backward line)

Syntax: void bline(q)

Configuration q;

Description:

The argument q = (p, θ, κ) specifies a straight line or a circular arc. Basically the robot is supposed to follow this directed path element. The robot will track the line q until it passes q itself and will transfer to the next path segment. If there is no next path segment, the robot will start to slow down at the configuration q and eventually stop with the current acceleration rate.

Precisely speaking, the robot leaves the segment q when the robot's image reaches q (or is downstream of q). (The image of a point P on a path π is defined as the closest point on π from P)

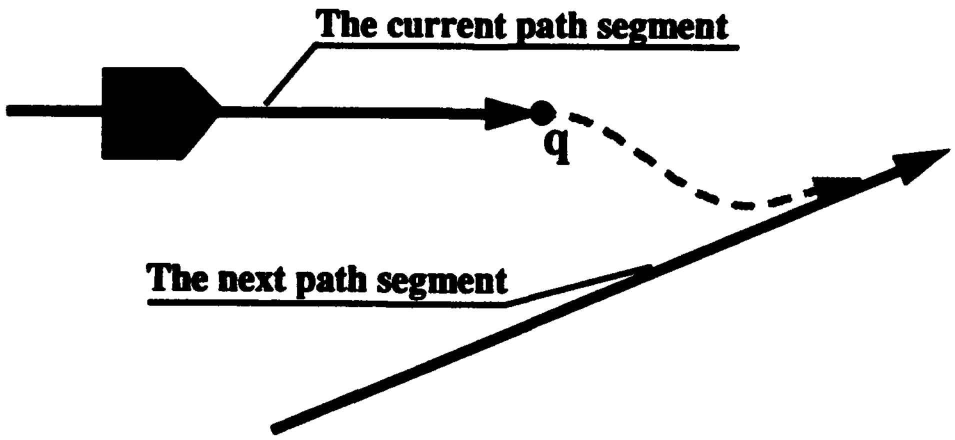


Figure 9: The bline function.

4. Skip

Syntax: void skip()

Description:

When the robot reads a *skip* function, it will immediately leave the current path segment and will track the next one. All the motion functions called before the *skip* function will be discarded and the path function which follows it will be tracked hereafter (Figure 10). In normal usage, a *skip* function comes with some tests.

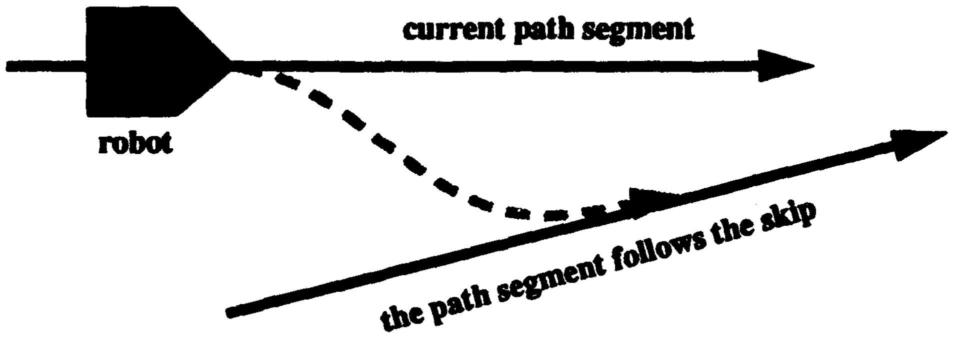


Figure 10: The skip function.

Example:

For example, in an obstacle avoidance case, while the robot is tracking on L1, the robot's sonar continuously senses the distance to a possible obstacle in front. When the distance is less than 100 centimeter, a *skip* function will be called and the robot will immediately transfer to L2 (Figure 11). An example program in pseudocode for this behavior is:

```
{
line(L1);
while (sonar_dectect_distance >= 100);
skip();
line(L2);
}
```

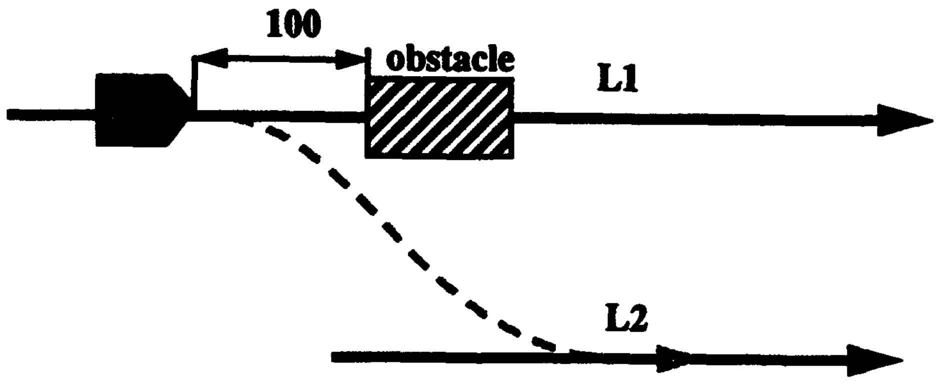


Figure 11: An application example for a skip function.

5. Stop

Syntax: void stop(q)

Configuration q;

Description:

The robot will track on the line q and stop at it. When the rest of the path is equal to $v^2 / 2a$, where v is the current speed and a is acceleration, it will start decelerating and will fully stop at q.

Precisely speaking, this deceleration process is controlled not by the robot's position itself but by its image on q, i.e., the *stop* function works in such a way that the robot's image will stop at q or if the robot's image is already downstream of q, it immediately decelerates.

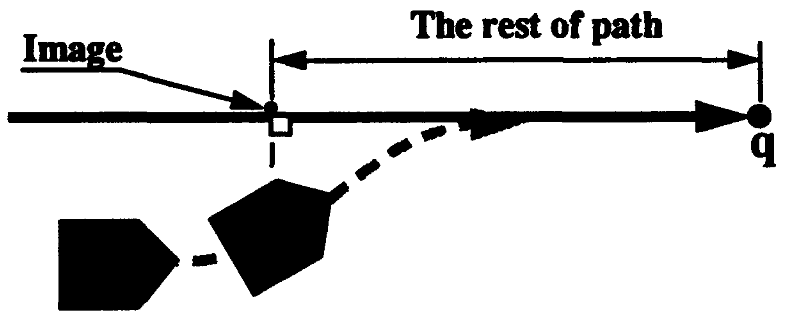


Figure 12: The stop function.

V. FUNCTIONAL ELEMENTS IN BACKGROUND PROCESS

There is a motion control program which is executed every 0.01 seconds in the background process. It is not necessary for users to know how motions are controlled and users will not be allowed to use these control functions. This motion control program smoothly and precisely controls the robot's motions. The architecture of these background control functions is shown in Figure 13. The notations used in this chapter are defined as followings:

X: x coordinate of the robot's position.

Y: y coordinate of the robot's position.

θ: the robot's current orientation.

κ: the robot's current curvature.

ΔR: left wheel traveling distance.

ΔL: right wheel traveling distance.

ΔS: the traveling distance of the robot.

Δq: the angular change of the robot.

U_v: commanded translational velocity of the robot.

U_ω: commanded rotational velocity of the robot.

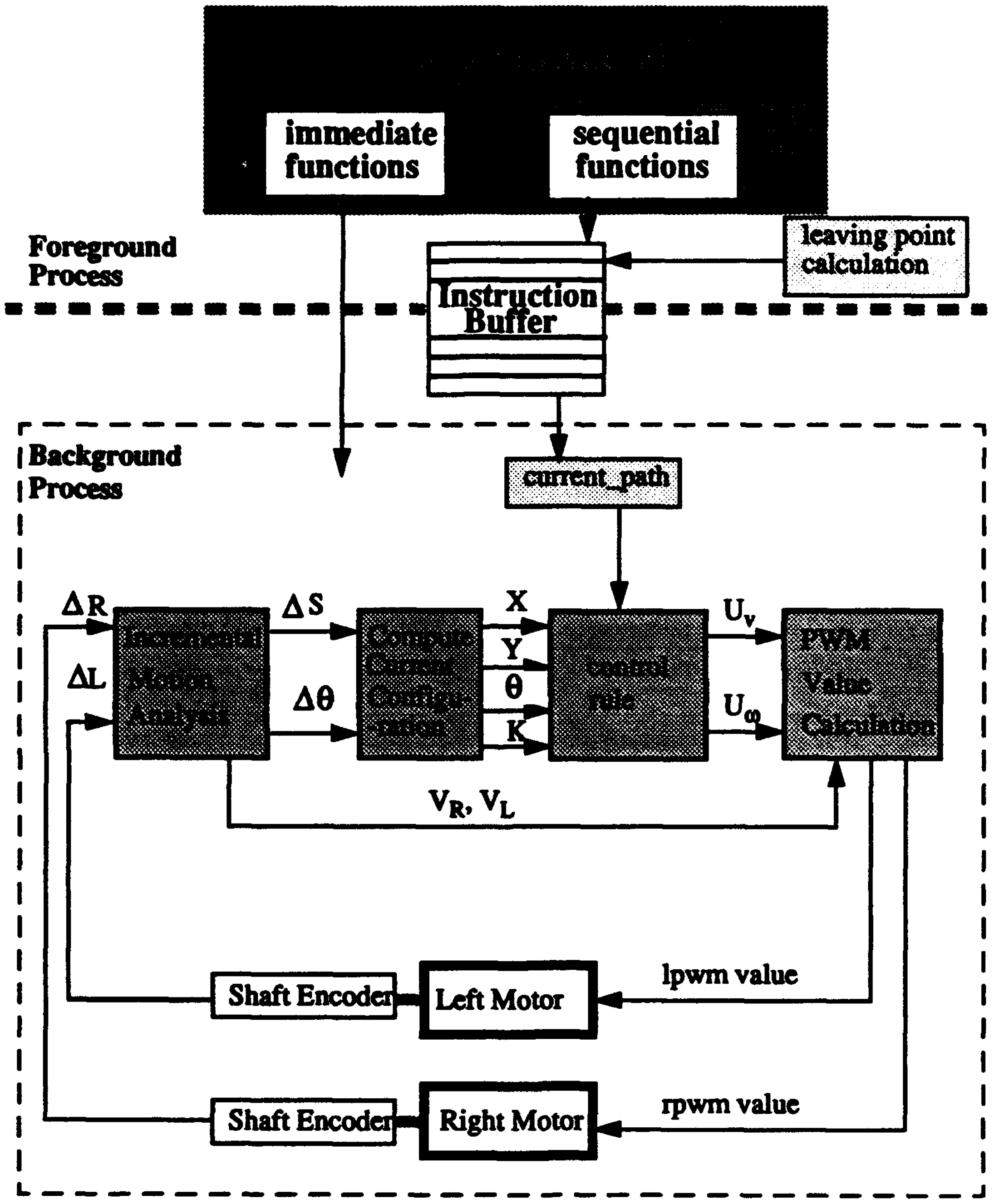


Figure 13: Background Motion Control Architecture.

A. INCREMENTAL MOTION ANALYSIS

The autonomous wheeled vehicle, 'Yamabico-11', has two motors to drive each of the two wheels separately. Each motor has a shaft encoder to sense the rotation of the motor's turning shaft. One complete shaft rotation gives five hundred and twelve pulses. Thus, the left and right shaft encoder can sense the actual amount of the motor's rotation. The travelling distance of the left and right wheels are computed through encoder readings, reduction gear box ratio (1:24) and wheel radius (10 cm) during the sampling time period of 0.01 seconds. The robot speed is obtained by dividing the distance with the sampling time. 'Yamabico-11' uses the differential drive method for steering. The difference between two wheels' distance divided by the robot's width (52.4 cm) is Δθ, the angle the robot has turned during the last sampling time. The travelling distance(ΔS), the rotational angle of the robot(Δθ) and both wheel's velocities are calculated by the following equations. Figure 14 illustrates their relationships with the robot.

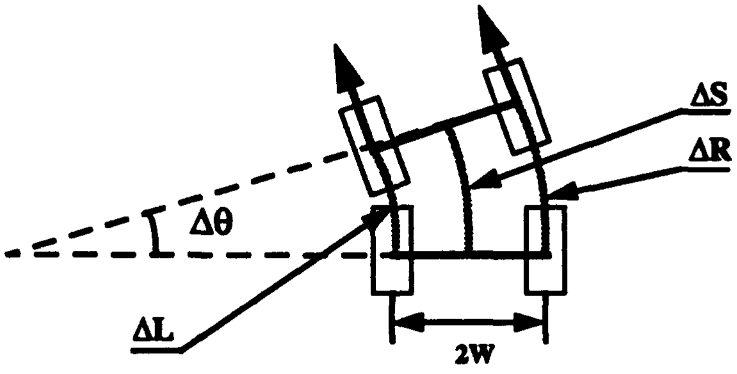


Figure 14: The robot's ΔS and Δθ.

r = wheel radius = 10 cm

ΔL = LEFT_DISTANCE = N_l (left encoder readings) / 512 x (1 / 24) x 2 π r

ΔR = RIGHT_DISTANCE = N_r (right encoder readings) / 512 x (1 / 24) x 2 π r

$\Delta S = (\Delta R + \Delta L) / 2$

ΔDISTANCE = $\Delta R - \Delta L$

2W= robot width = 52.4cm

$\Delta\theta$ = ΔDISTANCE / 2W

VELOCITY = ΔS / 0.01 second

The precise control of two drive wheels will provide the robot desired orientations (heading directions) and positions. It will let the robot follow the desired tracking path. All different kinds of motions of 'Yamabico-11', such as line to line, line to circle, circle to circle path tracking can be accomplished by this type of motion control.

By the robot's odometry sensor informations, we could let the robot know how far it has travelled and what its orientation is at any moment in time. After the robot gets these data from the odometry-sensor-reading function, the current-configuration-computation function will continuously compare the robot's current configuration (position and orientation) with the robot's designated path and keep the robot always on the correct tracking path. This path tracking capability will be included inside the commanded-speed-computation function and will be described later in this chapter.

B. CURRENT CONFIGURATION COMPUTATION

As mentioned in section A of this chapter, in order to let the robot always keep moving on the desired path, the correct robot's current configuration (position and orientation) will be very important information. After the odometry sensor reading function is executed, the robot's travel distance(ΔS) and the robot's rotational angle change($\Delta\theta$) are known. By using this data, the robot's current position, orientation and its kappa can be computed as following:

Our implementation robot of 'Yamabico-11' is a two dimensional autonomous mobile robot. The two dimensional Cartesian coordinate with X and Y axises has been used to represent the robot's position.

Let the robot's original position be (X_0,Y_0) and the robot's new position be (X_1,Y_1). From the following picture Figure 15, we know $X_1 = X_0 + \Delta X$, $Y_1 = Y_0 + \Delta Y$. The equations for calculating ΔX, $\Delta Y$are:

$\Delta X = \sin(\theta+(\Delta\theta/2)) * d$

$\Delta Y = \sin(\theta+(\Delta\theta/2)) * d$

$d = \Delta S$, if $\Delta\theta = 0$.

$d = \Delta S * \sin(\Delta\theta/2) / (\Delta\theta/2)$, if $\Delta\theta \neq 0$.

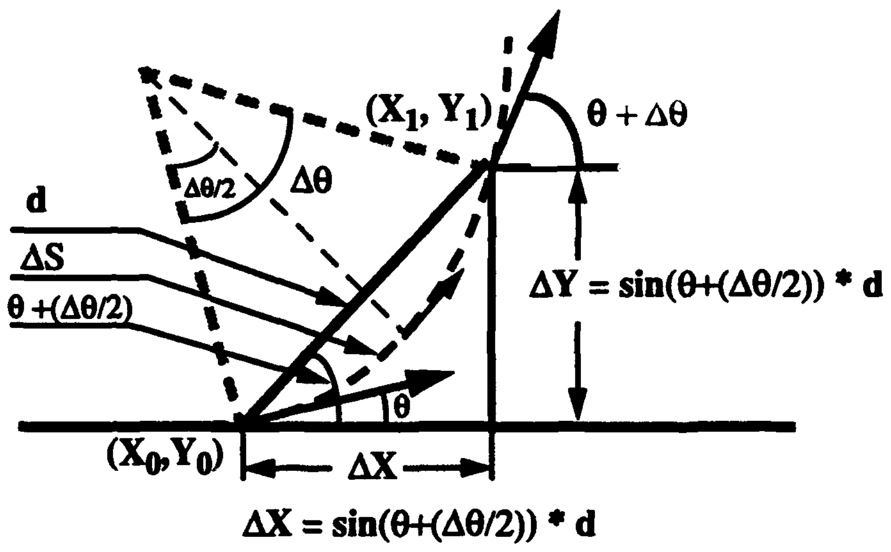

(X_1, Y_1)
$\theta + \Delta\theta$
$\Delta\theta/2$
$\Delta\theta$
d
ΔS
$\theta + (\Delta\theta/2)$
$\Delta Y = \sin(\theta+(\Delta\theta/2)) * d$
θ
(X_0,Y_0)
ΔX
$\Delta X = \sin(\theta+(\Delta\theta/2)) * d$

Figure 15: The relationship of robot's positions.

Figure 15, shows the robot's orientation change while the robot is tracking a curved path from position (X_0, Y_0) to another position (X_1, Y_1). Let the robot's old orientation be θ, then the robot's new orientation will be $\theta + \Delta\theta$.

The robot's motion control theory is the steering function. For understanding the control theory, a conventional notation **kappa** (κ), must be introduced. When the robot tracks on an arbitrary curve, the robot's state can be represented by three parameters, position, orientation and curvature. The curvature, κ, is defined by $d\theta / dS$ where the $d\theta$ is the robot's orientation change, the dS is the robot's traveling distance within one unit of time. The theory of the steering function is to control the robot's motion by changing its curvature.

C. CONTROL RULE

After the current-configuration-computation function has been executed, the robot gets enough information to know its current state. The steering function will be executed to get the robot its commanded speed. The *steering function* will use the path-tracking theory [Ref. 1] to control the robot's motion. Figure 16 shows the functional steps inside this program.

1. Steering Control

Before we describe the details of the steering function and path-tracking theory, a notation **image** needs to be defined first. The robot's image is defined as the projection from the robot's current configuration to the robot's goal path. Figure 17 shows the robot's images.

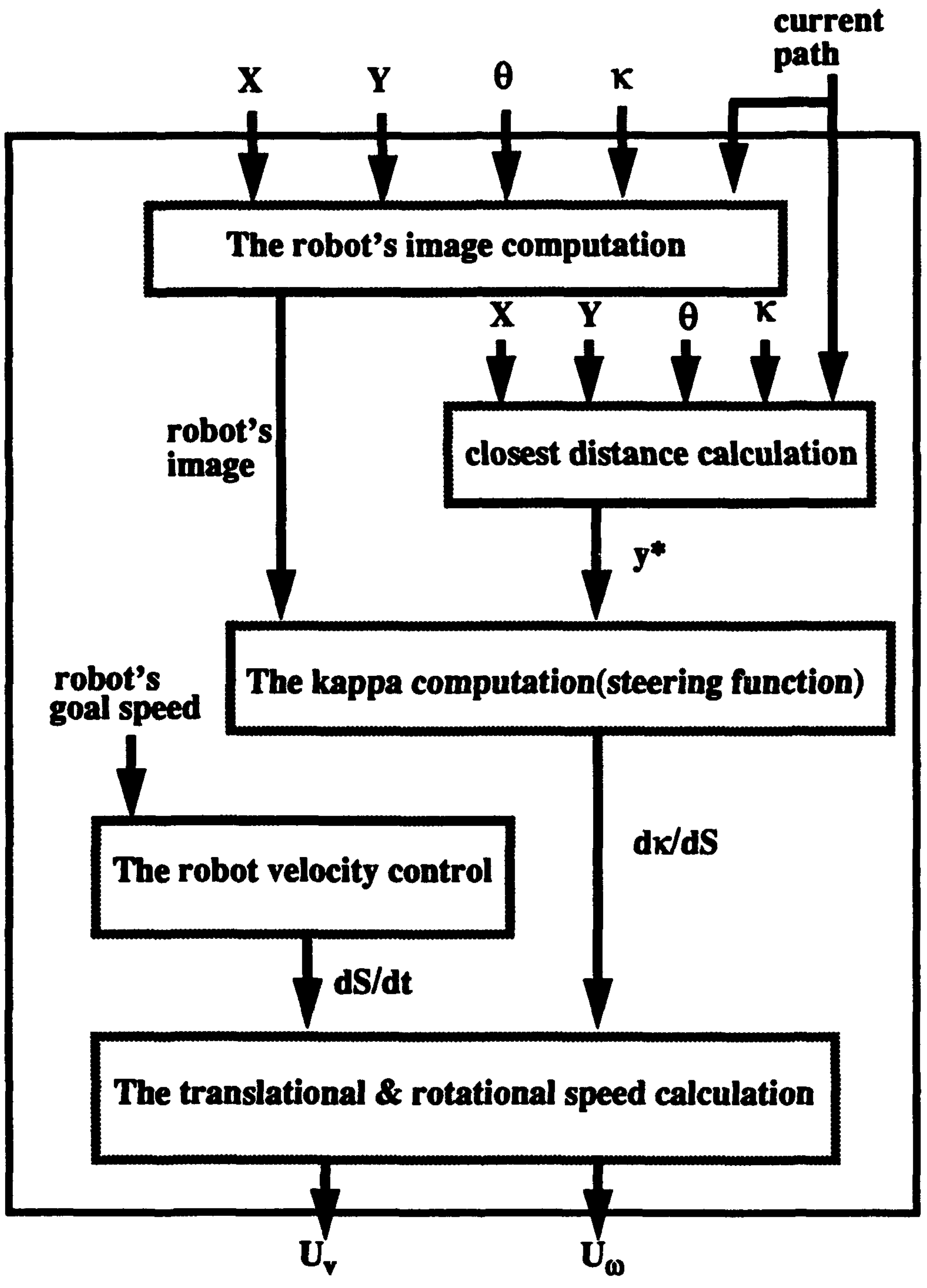

current path
X
Y
θ
κ
The robot's image computation
X
Y
θ
κ
robot's image
closest distance calculation
y*
robot's goal speed
The kappa computation(steering function)
dκ/dS
The robot velocity control
dS/dt
The translational & rotational speed calculation
U_v
U_ω

Figure 16: The control rule module.

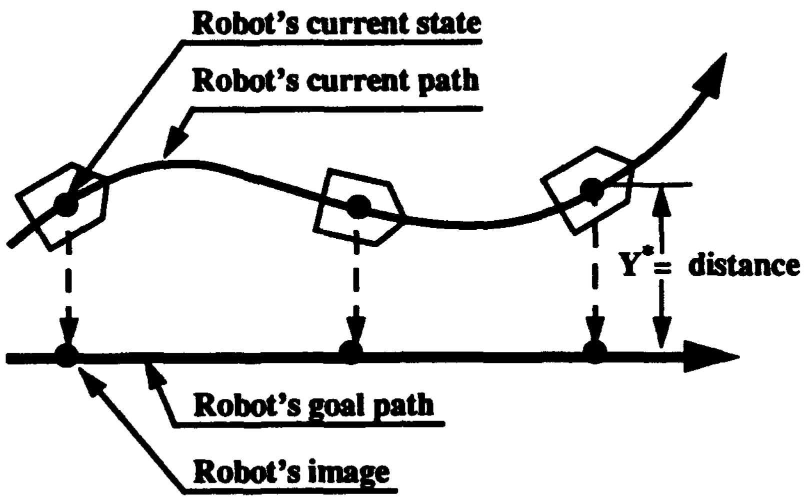


Figure 17: The robot's images.

The method to control our mobile robot is by changing the robot's curvature. The steering function use the path-tracking theory [Ref. 1] to compute the derivative of curvature. The equation is defined as:

$d\kappa/dS = -(a\Delta\kappa + b\Delta\theta + cY^*)$ --------- the equation of steering function.

and,

$a = 3K$, where K is a positive constant.

$b = 3K^2$

$c = K^3$

$\Delta\kappa$ = robot's kappa - image's kappa

Y^* = signed distance between robot's current position and its image.

In order to check the path-tracking algorithm and the steering function, a simulation program in LISP had been written to simulate the robot's path tracking control.The result is very good, Figure 18 to Figure 20 shows the robot can use the steering function to do line-to-line, line-to-circle, circle-to-circle tracking motions. The LISP program code will be provided in Appendix A.

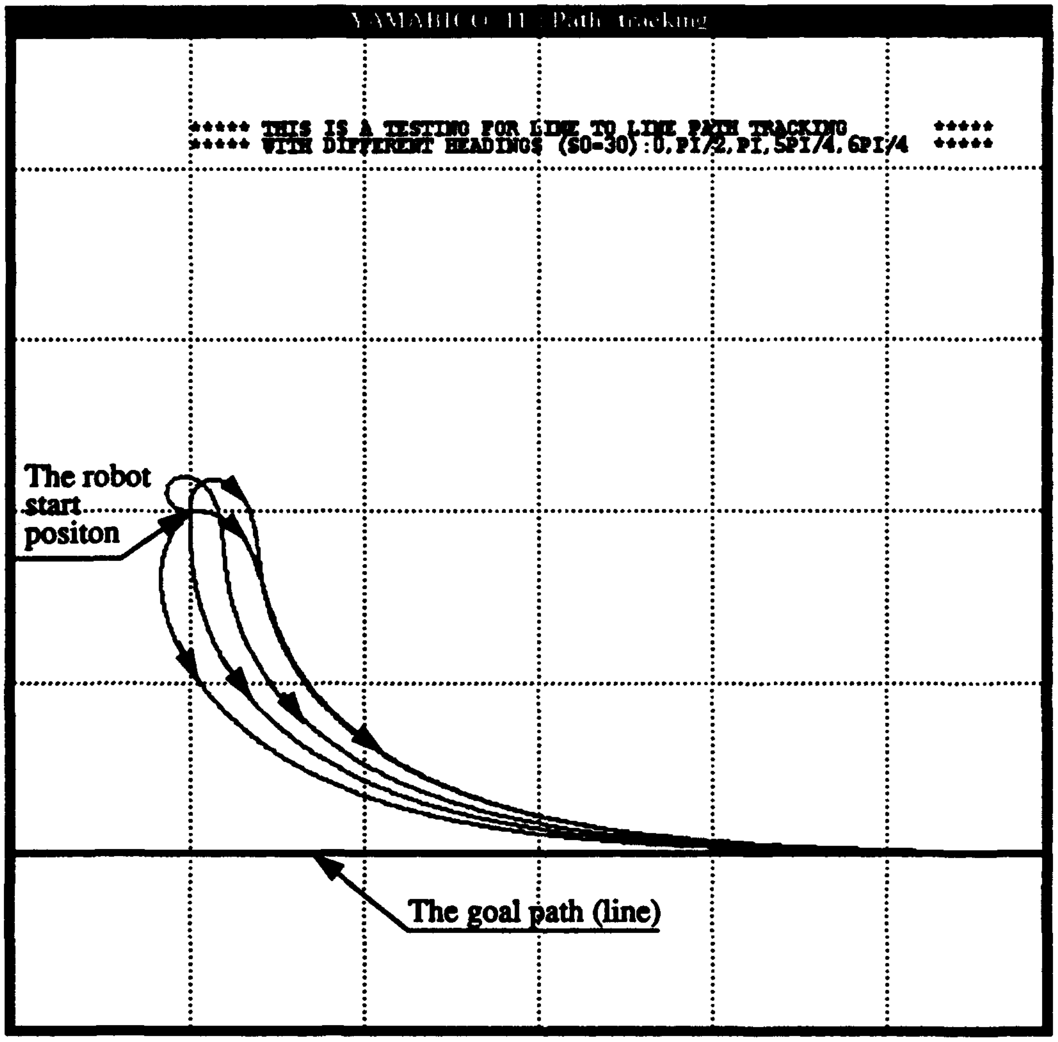
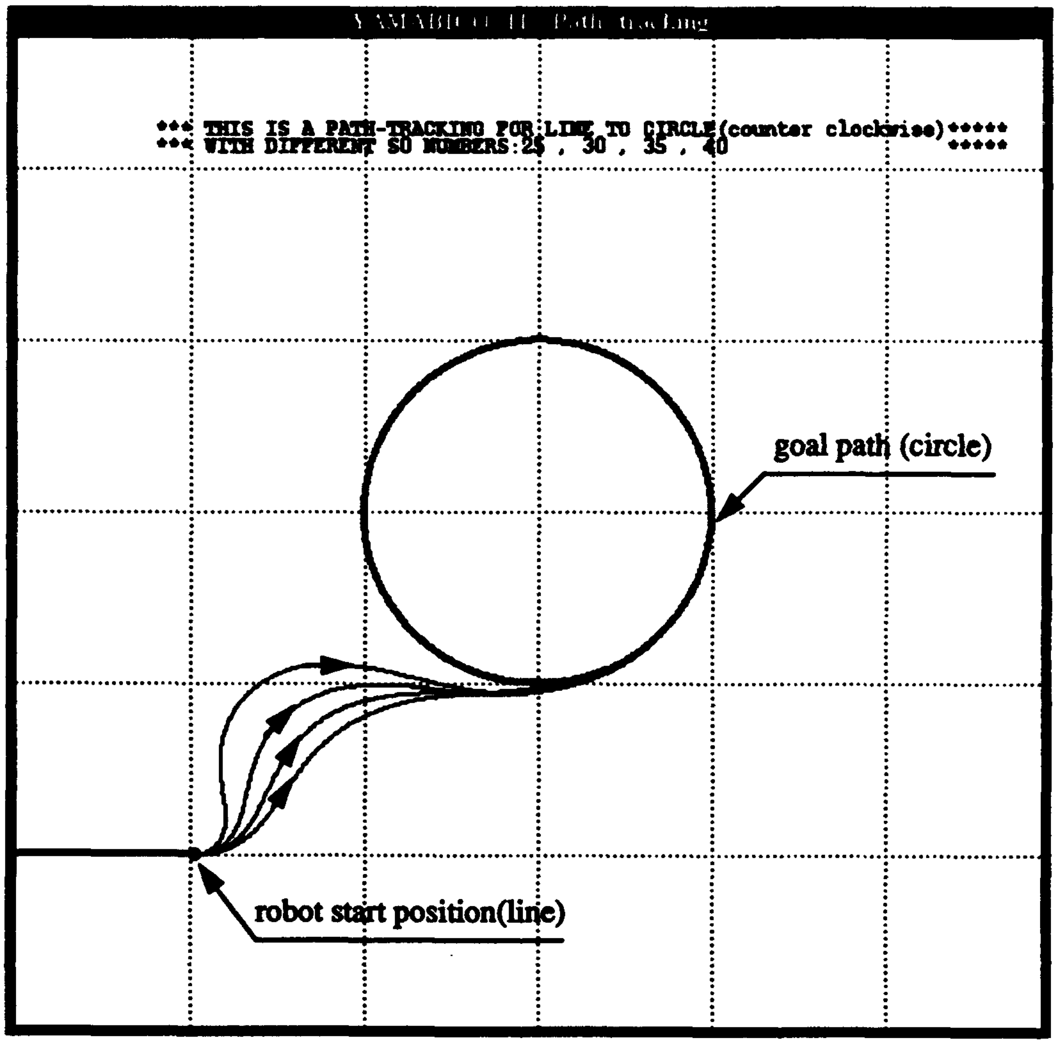
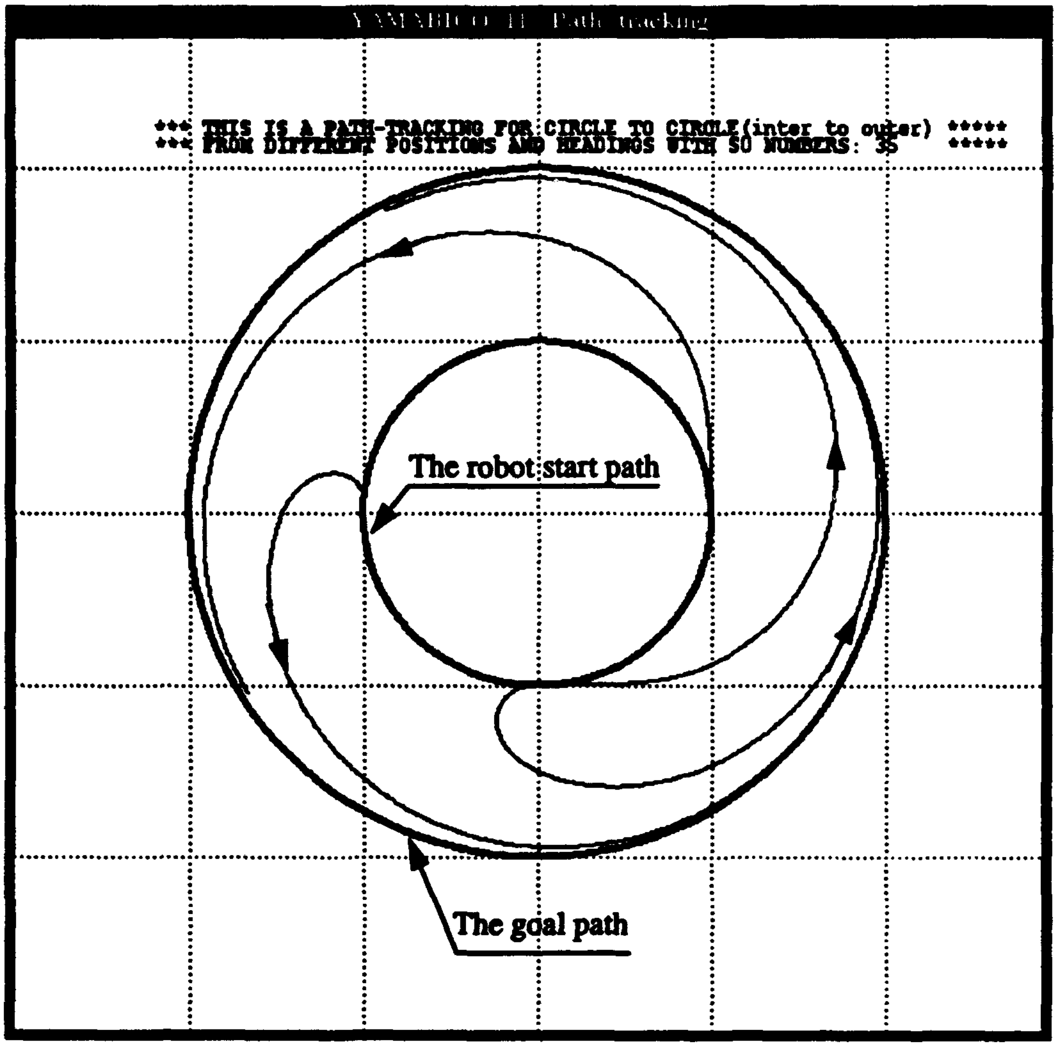

```
***** THIS IS A TESTING FOR LINE TO LINE PATH TRACKING          *****
***** WITH DIFFERENT HEADINGS (S0-30):0,PI/2,PI,5PI/4,6PI/4     *****
```

Figure 18: The line to line path tracking simulation.



```
*** THIS IS A PATH-TRACKING FOR LINE TO CIRCLE(counter clockwise)*****
*** WITH DIFFERENT S0 NUMBERS:25 , 30 , 35 , 40                  *****
```

Figure 19: The line to circle path tracking simulation.



```
*** THIS IS A PATH-TRACKING FOR CIRCLE TO CIRCLE(inter to outer) *****
*** FROM DIFFERENT POSITIONS AND HEADINGS WITH 50 NUMBERS: 35     *****
```

Figure 20: The circle to circle path tracking simulation.

2. Speed Control

As mentioned before, inside the control rule function, the robot's current configuration will be used to calculate the robot's image. The steering function will be used to compute the kappa. The velocity control function will take the commanded speed which can be set by user program and use the kappa to compute the robot's translational and rotational speed.

D. PWM VALUE CALCULATION

After computing the commanded translational and rotational speeds, the robot will need to use these data to compute the right wheel and left wheel speeds so that the robot will get the correct commanded speed. The robot's PWM value calculation function was designed to do this kind of job.

1. Inverse Kinematics

The commanded rotational and translational speeds which have been computed in control rule function are the desired speed of the robot. In the case of two-wheel-driving-control mobile robot, the separated speed calculation for each wheel is needed. The physical relationship between both wheels and the robot's speed is (Figure 21):

ω = robot's rotational speed

V = robot's translational speed

$2W$ = robot width

V_r = right wheel speed = $V + W * \omega$

V_l = left wheel speed = $V - W * \omega$

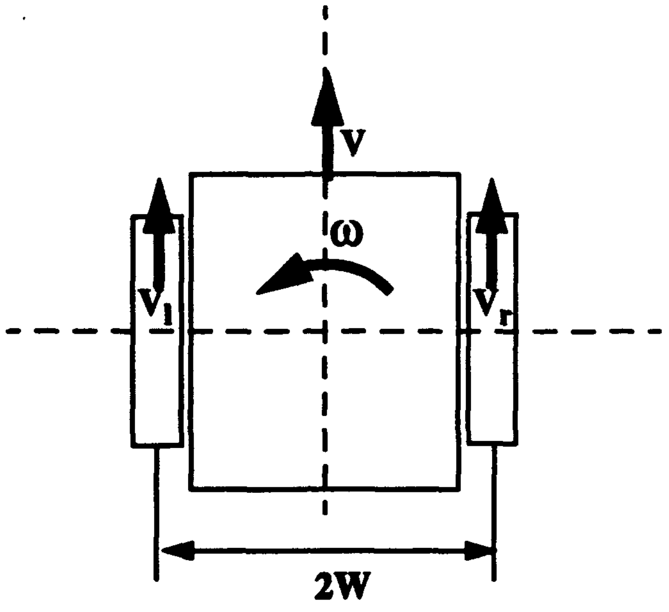


Figure 21: The relation between wheels and the robot's body.

2. The PWM Values

After each desired wheel speed are calculated, the last step for the robot's motion control is to convert the wheel speed to the motor control signals. The PWM value in 'Yamabico-11' is the physical control signals for the driving motors. The PWM value is defined as an inter range from -127 to 127. A PWM value divided by 128 is equal to the duty ratio of motor currents. The larger the absolute number, the more time will be activated to the motor. The zero PWM value will let the motor have a free state which means there is no any electrical currents to the motor. The absolute value of 127 will give the motor the largest electrical power which means the motor will get the longest activating time. The positive or negative number will turn the motor clockwise or counterclockwise. By experimental results (those experiments will be described in Chapter VII), a very reliable PWM value table is been provided in the motion control program (the code will be shown in Appendix B). By giving the wheel speed, the PWM table function will output a PWM value which guarantees the desired wheel speed.

VI. EXEPERIMENTAL RESULTS

In Yamabico-11, there are six wheels in total. Two wheels, lying on the robot's center line, drive the robot. The remaining four wheels are smaller than the drive wheels and each has a shock absorber connected to the robot's chassis, two in the front and two in the rear. Two DC motors are used to drive the wheels, one for each drive wheel. There is a reduction gear box connected between the motor and the drive wheel. Each drive wheel can be controlled separately and all the robot's motion control theories are implemented by successfully controlling each drive wheel.

The motor control board controls the motors. In Yamabico control program, the MML system, there is an integer variable called 'pwm'. The pwm value is ranged from -127 to 127, its absolute value represents the amount of time that the motor will be activated and its sign represents the motor rotational direction. The positive sign is clockwise for right wheel control value (rpwm) and is counterclockwise for left wheel control value (lpwm). The negative sign is just opposite. Each motor is activated by the amount of time which the pwm value represents. If we want the motor get half of the motor's maximum rotational speed, we will set the pwm value to 64 or -64 and if we want the motor get all rotational speed we will set it to 127 or -127.

The real relationship between the pwm value and the actual robot velocity are obtained by experiments. Figure 22 shows the relationship between the pwm value and the robot's forward speed when the motor's control frequency is 7.9 KHZ. Figure 23 shows the curve of the pwm value and the robot's backward speed at the 7.9 KHZ. These experimental programs are provided in Appendix C.

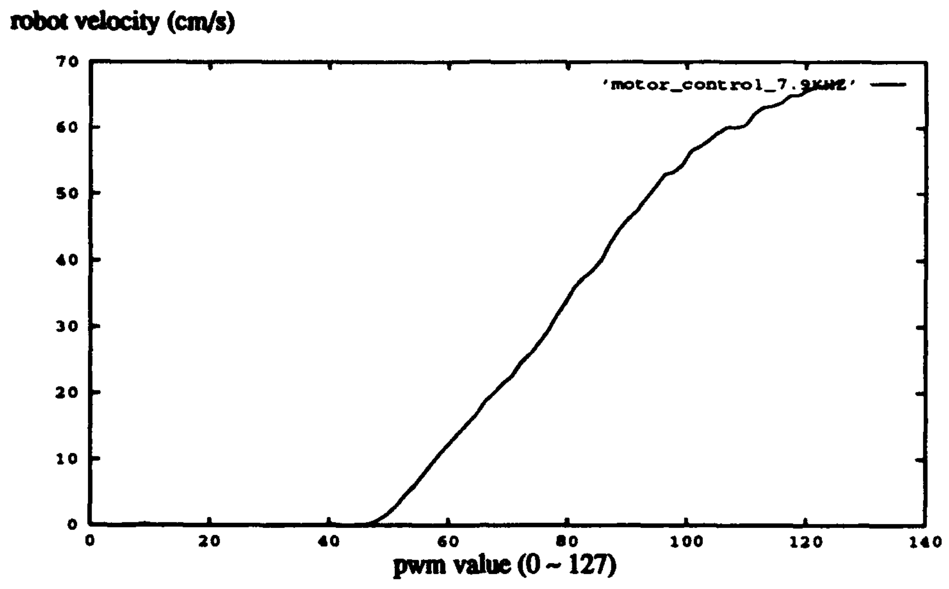


Figure 22: pwm value - forward speed curve of 7.9KHZ.

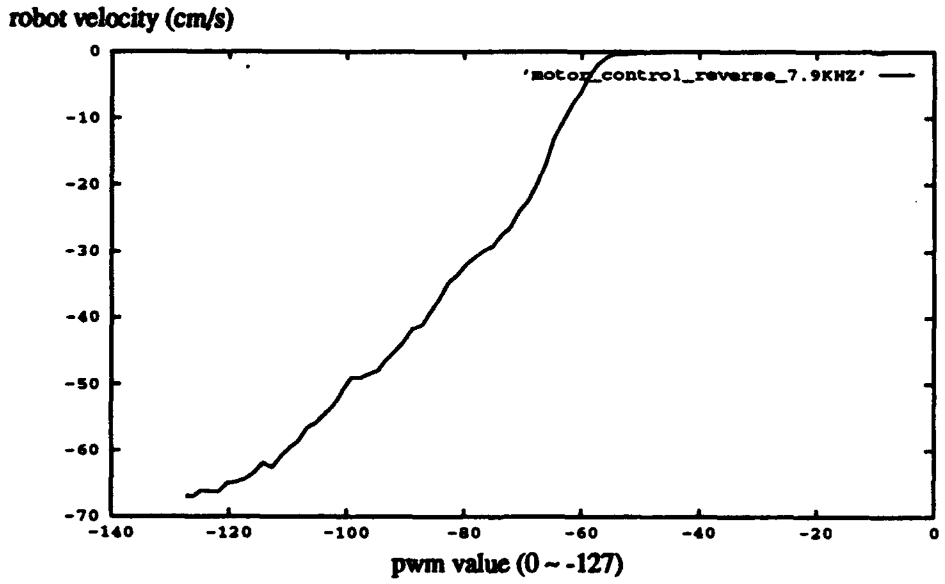


Figure 23: pwm value- backward velocity curve of 7.9KHZ.

These curves show that the robot starts to move forward at about a pwm value of 50 to overcome the robot's friction and starts to move backward at about -55. The robot's maximum velocity is about 67 cm/s at the pwm value of 127 or -127. From these curves, a precise motor control table is established. In Yamabico-11's motion control program, a function named 'pwm_lookup' has been written for this purpose (see Appendix B). Figure 24 shows the curves are different while the robot's speed is increasing or decreasing. For the same robot speed, the pwm value required in the increasing stage is about 10 more than that which is required in the decreasing stage.

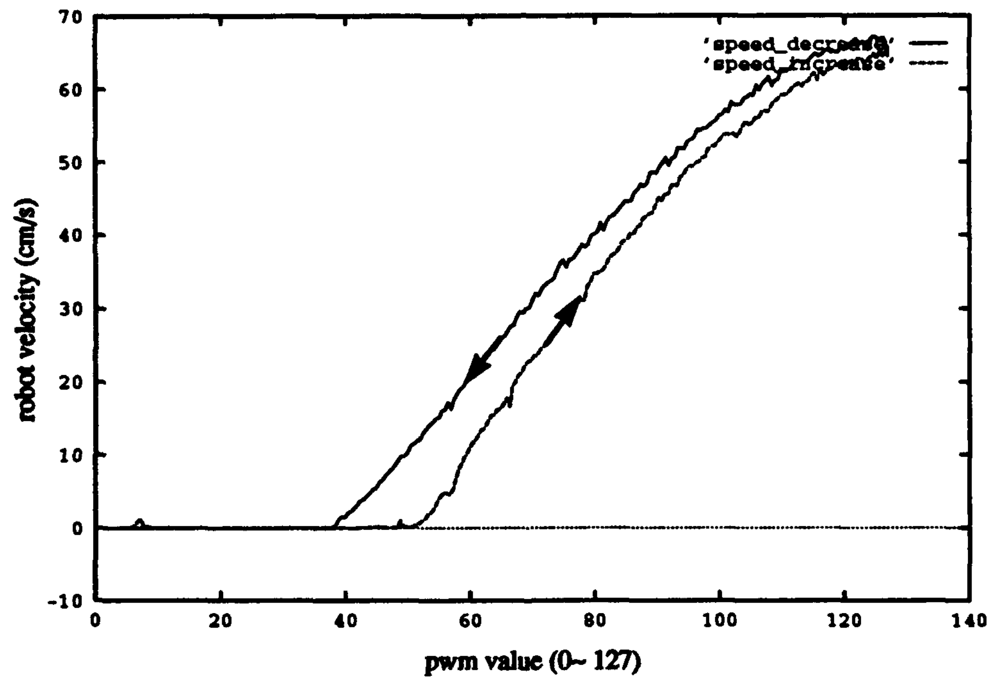


Figure 24: pwm-velocity curve for increasing and decreasing.

After the motor control table has been established, the relationship between robot's actual velocity and commanded speed needs to be checked. Figure 25 shows this relationship.The relationship between them is almost a forty-five-degree straight line. This result shows that the robot control program will allow the robot to achieve a velocity which is exactly the same as the commanded velocity. This capability is the very basis of wheeled robot. The program of this test is provided in Appendix C.

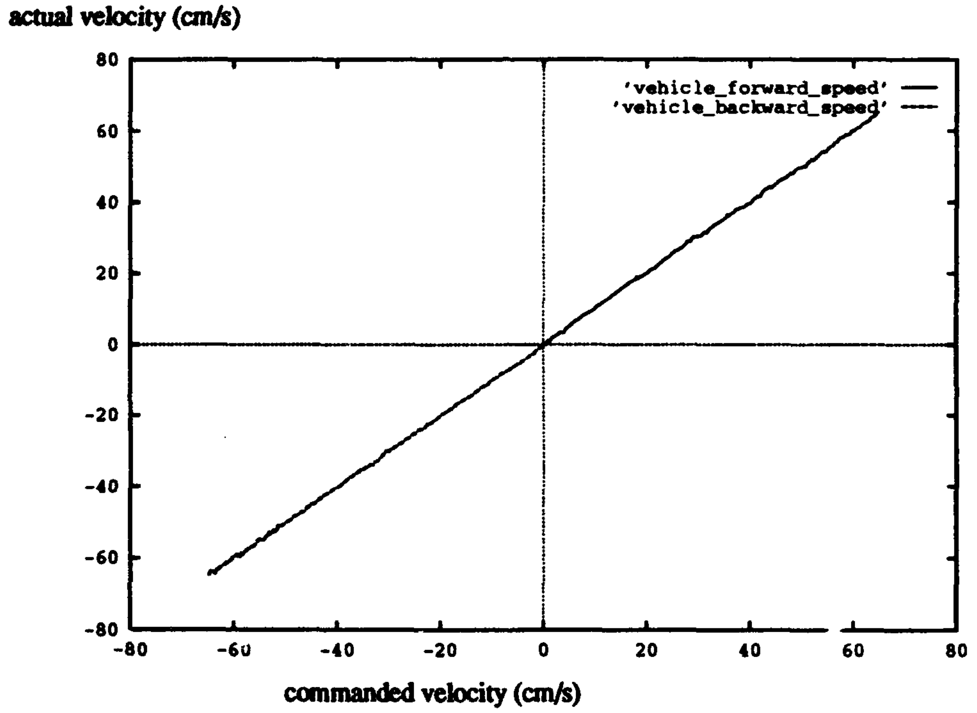


Figure 25: The actual speed-commanded speed curve.

VII. FINAL RESULT

Figure 26 shows the robot has successfully executed a line-circle-line mission. The robot's initial position was (0,-10) and its orientation was 0 degrees. The robot tracked on the line Y=-10 until it reached the leaving point. After this point, the robot tracked the circle until it reached another leaving point. The robot transitioned from circle to line Y=10 and stopped at point (0,10) with 180 degree of orientation. The user program is provided in Appendix D.

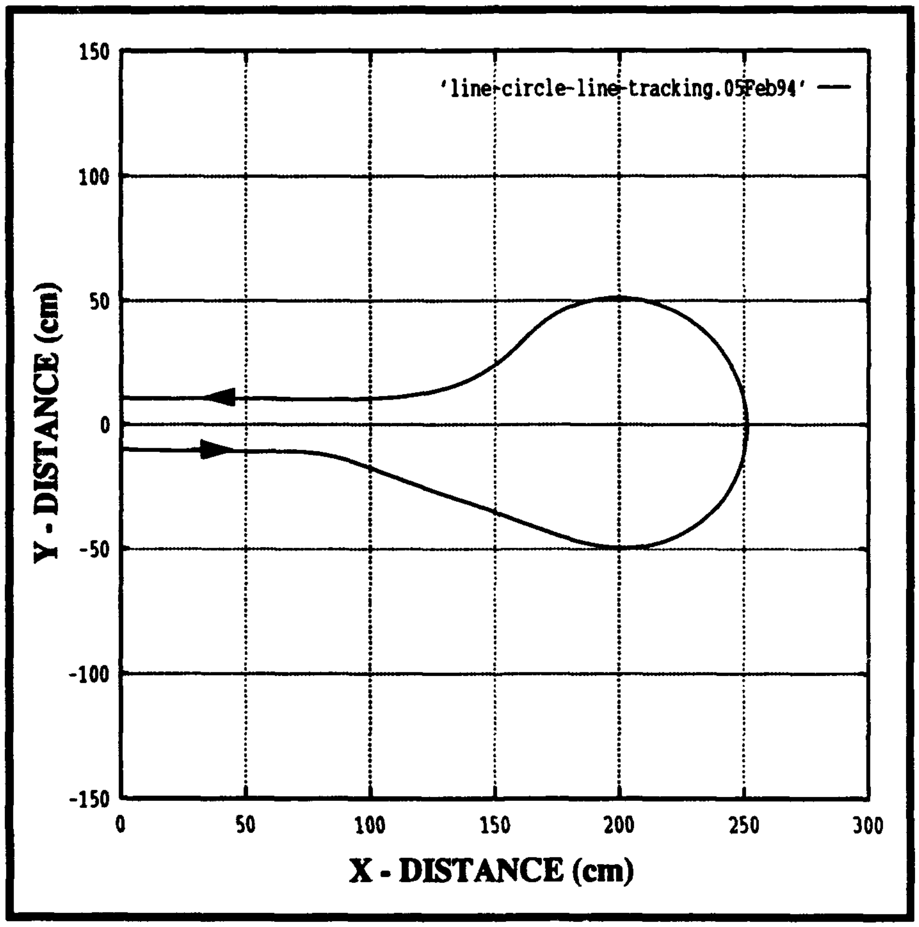


Figure 26: A line to circle and back to line motion.

Figure 27 shows the robot has executed a star-shaped motion mission, where the robot has made some sharp and safe turns when the robot transitioned from one line to another. This mission shows the robot's ability of tracking five lines with different orientations. The stable and safe turning capability has been demonstrated in this mission. The user program is provided in Appendix D.

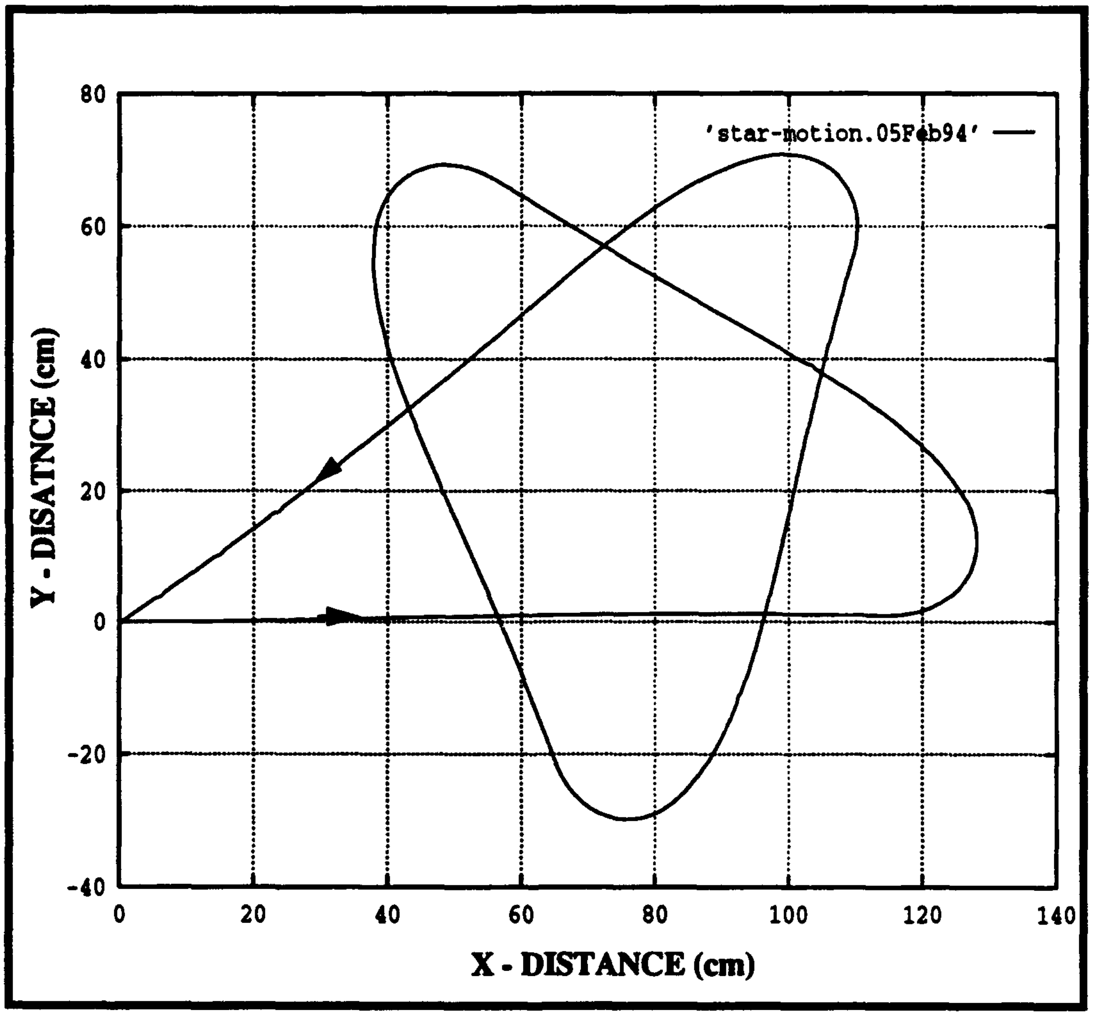


Figure 27: The Star-Shaped Motion.

The implementation of a path-tracking mission mentioned in Chapter I is a combination of several path elements. Figure 28 shows the robot has successfully executed the path-tracking mission. The robot starts at point (0,0), then tracks on line Y = 50, when the robot reaches to the optimum leaving point for transferring to line X = 200, the robot starts to transfer to line X = 200 and tracks on this line until it reaches the specified point (200,150), then the robot starts to transfer to its final path element, line X = 250, and then precisely stop at the destination, point (250,300). The user program is provided in Appendix D.

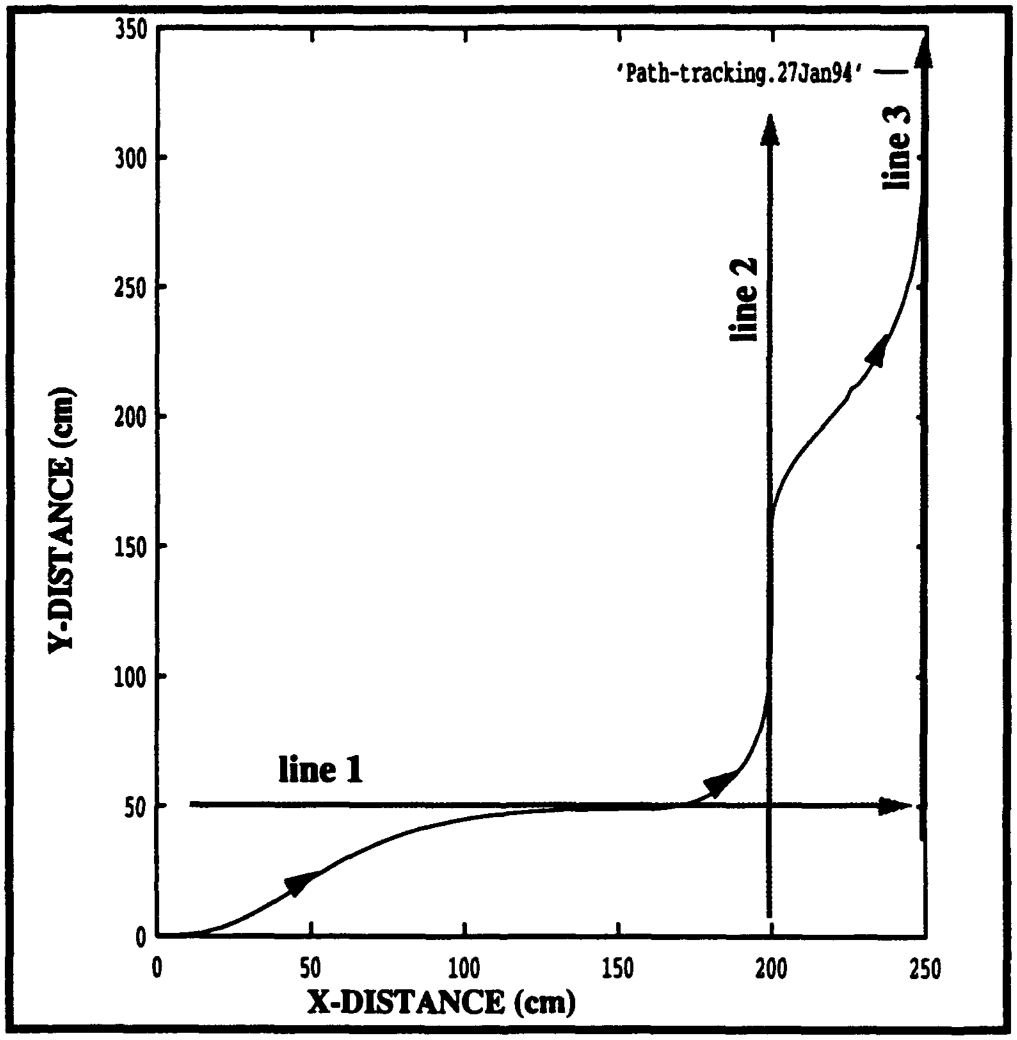


Figure 28: The robot executes a path-tracking mission.

'Yamabico-11' can use sonar as its environmental sensor to execute some obstacle avoidance missions. Figure 29 shows the robot has tracked a line Y = 0 with a goal configuration (500,0) and 0 degree orientation. The robot opens its front sonar while it is tracking on its current path, as soon as the distance from an obstacle is less than 100 centimeter, it transitions to an avoidance path which is line Y = -100 and opens the side sonar to detect the obstacle until it passes the obstacle. When the robot passes the obstacle, it returns to its original path and stops at its final goal configuration. This user program is in Appendix D.

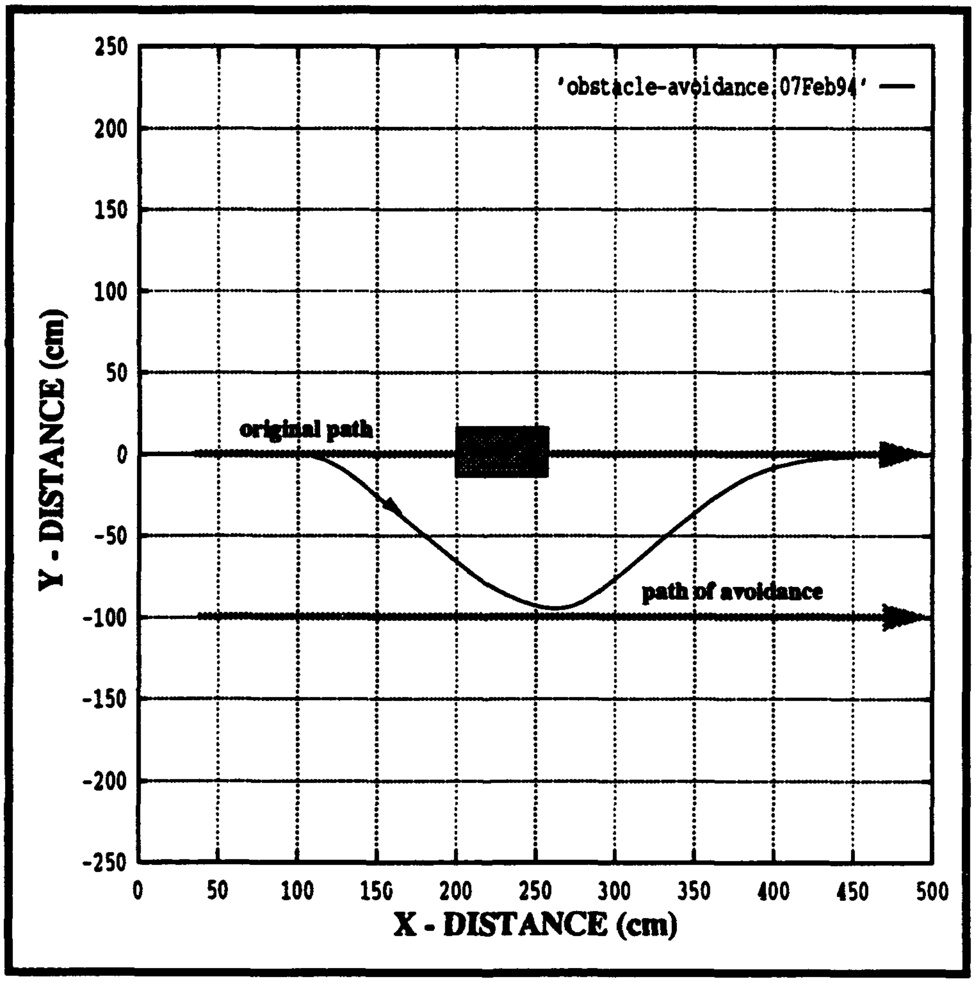


Figure 29: Obstacle avoidance mission.

APPENDIX

A. LISP PROGRAM FOR PATH TRACKING SIMULATION.

```
;*************************************************************
; This is main program 'path-tracking()'
;*************************************************************
(defun path-tracking (delta-length vehicle path distance-constant)
 (let*((const-k (/ 1 distance-constant))
      (A (* 3 const-k))
      (B (* 3  const-k const-k))
      (C (* const-k const-k const-k))
      (closest-distance 1000)
      (theta-difference 1000)
      (image)
      (differential-kappa)
      (delta-kappa)
      (current-vehicle))
  (do ((length 0 (setf length (+ length delta-length))))
      ((or (and (< (abs closest-distance) 0.5)  (< theta-difference 0.1))
           (> length 600.0))
       'Path-tracking-finished)
      (setf image (update-image vehicle path))
      (setf closest-distance (update-closest-distance vehicle path))
      (setf differential-kappa (- (+ (* A (- (nth 3 vehicle) (nth 3 image)))
                                     (* B (norm (- (nth 2 vehicle) (nth 2 image))))
                                     (* C closest-distance))))
      (setf delta-kappa (* differential-kappa delta-length))
      (setf current-vehicle vehicle)
(setf vehicle
   (compute-next-configuration vehicle delta-kappa delta-length))
(setf theta-difference (abs (- (nth 2 vehicle) (nth 2 image))))
(cw:draw-line (camera-window my-camera)
       (cw:make-position :x (first current-vehicle)
                :y (second  vehicle))
       (cw:make-position :x (first current-vehicle)
                :y (second  vehicle))
       :brush-width 0))))
```

```
;*************************************************************
; This is updat-image()
;*************************************************************
(defun update-image (vehicle path)
 (cond ((= (nth 3 path) 0.0)
     (let ((closest-distance (- (* (- (nth 1 vehicle) (nth 1 path))  (cos (nth 2 path)))
                                  (* (- (nth 0 vehicle) (nth 0 path))  (sin (nth 2 path))))))
       (list (+ (nth 0 vehicle) (* closest-distance (sin (nth 2 path))))
          (- (nth 1 vehicle) (* closest-distance (cos (nth 2 path))))
          (nth 2 path)
          (nth 3 path))))
     (t (let* ((radius (/ 1.0 (nth 3 path)))
          (origin-x (- (nth 0 path)
                 (*  radius (sin (nth 2 path)))))
          (origin-y (+ (nth 1 path)
                 (*  radius (cos (nth 2 path)))))
          (gamma (atan (- (nth 1 vehicle) origin-y)
                 (- (nth 0 vehicle) origin-x))))
          (list  (+ origin-x (* (abs radius) (cos gamma)))
              (+ origin-y (* (abs radius) (sin gamma)))
              (norm (+ gamma (* (/ PI 2)
                  (/ (nth 3 path)(abs(nth 3 path))))))
            (nth 3 path))))))
```

```
;*************************************************************
; This is compute-next-configuration()
;*************************************************************
(defun compute-next-configuration (vehicle delta-kappa delta-length)
  (let* ((new-kappa (+ (nth 3 vehicle) delta-kappa))
         (delta-theta (* delta-length new-kappa))
         (delta-l (* delta-length (/ (sin (/ delta-theta 2)) (/ delta-theta 2)))))
        (list (+ (nth 0 vehicle)
                 (* delta-l (cos (+ (nth 2 vehicle)
                                    (/ delta-theta 2)))))
              (+ (nth 1 vehicle)
                 (* delta-l (sin (+ (nth 2 vehicle)
                                    (/ delta-theta 2)))))
              (+ (nth 2 vehicle) delta-theta)
              new-kappa)))

;*************************************************************
; This is norm()
;*************************************************************
(defun norm (angle)
  (cond ((> angle PI)
         (- angle (* 2 PI)))
        ((< angle (- PI))
         (+ angle (* 2 PI)))
        (t angle)))
```

```
;*************************************************************
; This is update-closest-distance ()
;*************************************************************

(defun update-closest-distance (vehicle path)
  (/ (- (* (- (- (nth 0 vehicle) (nth  0 path)))
           (+ (* (nth 3 path) (- (nth 0 vehicle) (nth  0 path)))
             (* 2 (sin (nth 2 path)))))
        (* (- (nth 1 vehicle) (nth  1 path))
           (- (* (nth 3 path) (- (nth 1 vehicle) (nth  1 path)))
             (* 2 (cos (nth 2 path))))))
     (+ 1 (sqrt (+ (* (+ (* (nth 3 path) (- (nth 0 vehicle) (nth  0 path)))
                         (* 2 (sin (nth 2 path))))
                      (+ (* (nth 3 path) (- (nth 0 vehicle) (nth  0 path)))
                         (* 2 (sin (nth 2 path)))))
                    (* (+ (* (nth 3 path) (- (nth 1 vehicle) (nth  1 path)))
                         (* 2 (cos (nth 2 path))))
                      (+ (* (nth 3 path) (- (nth 1 vehicle) (nth  1 path)))
                         (* 2 (cos (nth 2 path))))))))))
```

```
;*************************************************************
; This is draw-grid ()
;*************************************************************

(defun draw-grid ()
  (draw-line-in-camera-window my-camera '(0. 100.) '(1000. 100.))
  (draw-line-in-camera-window my-camera '(0. 200.) '(1000. 200.))
  (draw-line-in-camera-window my-camera '(0. 300.) '(1000. 300.))
  (draw-line-in-camera-window my-camera '(0. 400.) '(1000. 400.))
  (draw-line-in-camera-window my-camera '(0. 500.) '(1000. 500.))
  (draw-line-in-camera-window my-camera '(100. 0.) '(100. 1000.))
  (draw-line-in-camera-window my-camera '(200. 0.) '(200. 1000.))
  (draw-line-in-camera-window my-camera '(300. 0.) '(300. 1000.))
  (draw-line-in-camera-window my-camera '(400. 0.) '(400. 1000.))
  (draw-line-in-camera-window my-camera '(500. 0.) '(500. 1000.)))
```

```
;************************************************************
; This is draw-circle ()
;************************************************************
(defun draw-circle (origin-x origin-y radius)
  (cw:draw-circle-xy (camera-window my-camera) origin-x origin-y radius
                     :brush-width 3))




;************************************************************
;  load other programs
;************************************************************

(load "rigid-body.cl")
(load "robot-kinematics.cl")
(load "camera.cl")
(setf my-camera (make-instance 'camera))
(create-camera-window my-camera)
```

B. MOTION.C

```
#include "mml.h"

/*********************************************************************
FUNCTION  : control()
PARAMETERS: none
PURPOSE   : Reads robot encoders to update odometry every 10 msec (INTVL) and
             then sends commands to the motors that drive the wheels.
RETURNS   : pwm commands to drive the left and right drive  wheels.
CALLED BY : motor (assembly language code)
CALLS     : evaluate_incremental_motion(), new_config(), store_loc_trace_data()
             process_set_rob0(), get_velocity(), simulator_new_config(),
             evaluate_pwm().
*********************************************************************/
long int control()
{

double vl, vr;

double delta_s, delta_theta;

void store_loc_trace_data();

void get_velocity();

#ifndef SIM  /* compute delta_s, delta_theta, and velocities of left, right wheels */

evaluate_incremental_motion(&delta_s,&delta_theta,&vl,&vr);

new_config(delta_s, delta_theta);  /* update current configuration */

store_loc_trace_data(vehicle.t, vehicle.k, uv, uw);

if (!motor_on)  /* if the vehicle's motors are not on then return */

return;

#endif

  process_set_rob0();   /* for set_rob0 function temporal exec */

  get_velocity(&uv, &uw); /*calculate the translational and rotational  velocities */
#ifdef SIM
 simulator_new_config(uv, uw); /* for simulator use only */
#endif
return evaluate_pwm(vl,vr);  /*  Calculate pmw */
}/* end control */
```

```
/**********************************************************************
FUNCTION  : evaluate_incremental_motion
PARAMETERS: delta_s, delta_theta, vl, vr
PURPOSE   : get encoder information  returns how far the left and right
            wheels have moved forward in one step
RETURNS   : pointers of delta_s, delta_theta, vl, vr
CALLED BY : control()
CALLS     : read_left_wheel_encoder(), read_right_wheel_encoder()
**********************************************************************/
evaluate_incremental_motion(delta_s,delta_theta,vl,vr)
double *delta_s,*delta_theta,*vl,*vr;
{

double delta_r, delta_l;
double read_left_wheel_encoder();
double read_right_wheel_encoder();

if (status != RMOVE)

tread = TREAD;    /* Narrower trend width for forward motion */

else

tread = TREAD_R;

delta_r = read_right_wheel_encoder();

delta_l = read_left_wheel_encoder();

(*delta_s) = (delta_r + delta_l) / 2.0;

ss += (*delta_s);

(*delta_theta) = (delta_r - delta_l) / tread;

(*vr) = delta_r / INTVL;  /* calculate left and right wheel velocity */

(*vl) = delta_l / INTVL;

}
```

```
/************************************************************************
FUNCTION:  new_config()
PARAMETERS: delta_s, delta_theta
PURPOSE:  Updates the robot's current configuration based upon
          the input values of delta_s and delta_theta.
RETURNS:  none
CALLED BY:  control()
CALLS:    none
COMMENTS:  19 Apr 93 - Dave MacPherson
TASK: Level 4 interrupt
************************************************************************/
void new_config(delta_s, delta_theta)
double delta_s, delta_theta;
{

double sinc;

double dtheta2 = delta_theta / 2.0;

sinc = delta_s;

if (delta_theta)

sinc *= sin(dtheta2) / dtheta2;

 /* Update The vehicle's odometry estimate  */
vehicle.x += sinc * cos(vehicle.t + dtheta2);

vehicle.y += sinc * sin(vehicle.t + dtheta2);

vehicle.t += delta_theta;

vehicle.k = kappa;

cur_x = vehicle.x;

cur_y = vehicle.y;

cur_t = vehicle.t;

} /* end new_config */
```

```
/*********************************************************************
FUNCTION  : process_set_rob0()
PARAMETERS: none
PURPOSE   : This function is for set_rob0 function temporal execution
RETURNS   : none
CALLED BY : control()
CALLS     : norm()
*********************************************************************/
process_set_rob0()
{

if (setting_configuration)
 {

 setting_configuration = NO;
 vehicle.x = set_P.x;
 vehicle.y = set_P.y;
 vehicle.t = norm(set_P.t);

 }

}
```

```
/*********************************************************************
FUNCTION:   get_velocity()
PARAMETERS: uv, uw
PURPOSE: Determines the robot velocity and rotational
            velocity based upon vel_c and kappa.
RETURNS:   *uv, *uw
CALLED BY: control()
CALLS:     commanded_velocity(), commanded_kappa(), transition_point_test()
TASK: Level 4 interrupt
*********************************************************************/
void get_velocity(uv, uw)
double *uv, *uw;
{
switch (status)
{
case SSTOP:

length_s = 0.0;

vel_c = 0.0;

if (wait_cnt == 0)
  {

  (*uv) = vel_c;

  (*uw) = 0.0;

  read_inst();

  } else

    wait_cnt--;
break;

case SLINE:

(*uv) = vel_c = commanded_velocity();/* commanded velocity */

(*uw) = commanded_kappa() * vel_c;/* commanded omega */
```

```
if(get_inst != put_inst)
  {

    if (transition_point_test(current_image, get_inst->tp))
     {
       --no_o_paths;

       current_robot_path.pc = get_inst->c;

       current_robot_path.type = get_inst->class;

       read_inst();
      }

  }
if (skip_flag_control)
  {

    current_robot_path.pc = get_inst->c;

    read_inst();

    skip_flag_control = 0;

   }

break;

case SBLINE:

(*uv) = vel_c = commanded_velocity();/* commanded velocity */

(*uw) = commanded_kappa() * vel_c;/* commanded omega */

if (vel_c < 0.5 && EU_DIS(current_image.x, current_image.y,
   current_robot_path.pc.x, current_robot_path.pc.y) < 4.0)
  {
   status = SSTOP;

   read_inst();

   }
```

```
if (skip_flag_control)
 {

  current_robot_path.pc = get_inst->c;

  read_inst();

  skip_flag_control = 0;

 }

break;

case SCONFIG:

(* uv) = vel_c = commanded_velocity();  /* commanded velocity */

current_image = update_cubic_image(vehicle,current_robot_path);


/* Now update the global kappa that is used to control the robot's actual motion */
(* uw) = update_cubic_kappa(vehicle, current_image) * vel_c;

     /* There are many tests to see if the end of the spiral
       has been reached, easiest is to compare image_s to
       precomputed length of spiral, stored in pp.x0 */
if (image_s > current_robot_path.pp.x0)
  {

     read_inst();

  } /* end if */

     break;

case SPARABOLA:

break;

case RMOVE:

(*uv) = 0.0;
```

```
(*uw) = rvel_c = get_rotational_vel();

break;

case SERROR:

vel_c = commanded_velocity();

if (vel_c <= VEL1)
 {
  vel_c = 0.0;

  motor_on = ON;

 }

break;

default:

(*uv) = 0.0;

(*uw) = 0.0;

break;

` /* end switch */

}/* end get_velocity() */
```

```
/**********************************************************************
FUNCTION  : simulator_new_config()
PARAMETERS: uv, uw
PURPOSE   : For the simulator compute vehicle's configuration based on left
            and right commanded wheel speed this is required in lieu of real
            odometry
RETURNS   :
CALLED BY : control()
**********************************************************************/
simulator_new_config(uv, uw)
double uv, uw;
{

double delta_theta, delta_dist1;

delta_theta = uw * INTVL;

delta_dist1 = uv * INTVL;

vehicle.x += (cos(vehicle.t + delta_theta / 2.0) * delta_dist1);

vehicle.y += (sin(vehicle.t + delta_theta / 2.0) * delta_dist1);

vehicle.t = vehicle.t + delta_theta;

vehicle.k = kappa;

}
```

```
/***********************************************************************
FUNCTION  : evaluate_pwm()
PARAMETERS: lpwm,rpwm,vr,vl
PURPOSE   : compute mcw and the pwm for left and right wheels
RETURNS   : pointers of lpwm, rpwm
CALLED BY : control()
CALLS     : pwm_lookup()
***********************************************************************/
long int evaluate_pwm(vl,vr)
double vl, vr;
{

 int bufpwm;

 int lpwm, rpwm, lpwm0, rpwm0;

 double uvl, uvr, delta_vl, delta_vr;

 double a = 0.7;

 double pwm_lookup();

 tread2 = 0.5 * tread; /* If robot is in the rotate mode use wider trend width */

 uvl = uv - tread2 * uw;      /* compute commanded left and */

 uvr = uv + tread2 * uw;      /* right wheel velocities     */

 delta_vl = uvl - vl;

 delta_vr = uvr - vr;

 /* adjust pwm's  based upon the difference between the
    calculated wheel velocity and the odometry wheel velocity */
 lpwm0 = pwm_lookup(uvl) + kpw_b * delta_vl; /* left wheel */

 rpwm0 = pwm_lookup(uvr) + kpw_b * delta_vr;  /* right wheel */

 lpwm = a * lpwm0 + (1.0 - a) * lpwm1;

 rpwm = a * rpwm0 + (1.0 - a) * rpwm1;
```

```
  lpwm1 = lpwm;

  rpwm1 = rpwm;

  #ifndef SIM

  if (mv_direction < 0.0)    /* set up motor control word (mcw) and
                                              threshold pwm values */
   {

     bufpwm = lpwm;

     lpwm = -rpwm;

     rpwm = -bufpwm;

   }

  mcw = (mcw & 0xf0f0) | (lpwm > 0?1:2) | (rpwm > 0 ? 0x0100 : 0x0200);

  if (lpwm > 127)
       lpwm = 127;

  else if (lpwm < -127)
       lpwm = -127;

  if (rpwm > 127)
     rpwm = 127;

  else if (rpwm < -127)
       rpwm = -127;

return (lpwm << 16 | rpwm & 0xff);

#endif

}
```

```
/***********************************************************************
FUNCTION:  commanded_velocity()
PARAMETERS: none
PURPOSE:   Determines the current robot translational velocity.
RETURNS:   double
CALLED BY: control()
CALLS:     rest_of_path()
TASK:      Level 4
***********************************************************************/
double commanded_velocity()
{

double vel_gg; /* temporary goal velocity */
double rest_of_path();

dvel = tacc * INTVL;

if (status == SBLINE &&
   2.0*tacc * rest_of_path(current_robot_path, current_image) <= vel_c * vel_c)
 {

   vel_c = max2(vel_c - dvel, 0.0);

 }
else
 {
   vel_gg = min2(vel_g, WHEEL_MAX / (1 + TREAD / 2 * fabs(kappa)));

   if (vel_gg >= vel_c)
   vel_c = min2(vel_c + dvel, vel_gg);

   else
   vel_c = max2(vel_c - dvel, vel_gg);

 }

delta_dist = INTVL * vel_c;

return vel_c;

} /* end commanded_velocity() */
```

```
/*********************************************************************
FUNCTION:  commanded_kappa
PARAMETERS: none
PURPOSE:   Main steering function for MML.
RETURNS:   void
CALLED BY: stepper, get_velocity()
CALLS:     limit(), update_image()
*********************************************************************/
double commanded_kappa()
{

  double delta_d;

  double dkappa1;

  double update_delta_d();

  double limit();

  current_image = update_image();

  delta_d = update_delta_d();

  dkappa1 =  -aa * (vehicle.k - current_image.k)
             -bb * (norm(vehicle.t - current_image.t))
             -cc * limit(delta_d);

  kappa = vehicle.k + dkappa1 * delta_dist;

  return kappa;

} /* commanded_kappa */
```

```
/**********************************************************************
FUNCTION:  update_image()
PARAMETERS: vehicle, current_robot_path
PURPOSE:   calculates the current_image for commanded_kappa()
RETURNS:   CONFIGURATION
CALLED BY:  commanded_kappa
CALLS:     sin, cos
**********************************************************************/
CONFIGURATION update_image()
{

 double radius, gamma, close_dist;

 POINT   origin;

 CONFIGURATION image;

 CONFIGURATION path;

 /* 10/19/93 shorten globle variable current_robot_path.pc */
 path = current_robot_path.pc;

 if (path.k == 0.0)
   {

      close_dist=(((vehicle.y - path.y) *cos(path.t)) - ((vehicle.x -path.x) *sin(path.t)));

    image.x = vehicle.x + close_dist * sin(path.t);

    image.y = vehicle.y - close_dist * cos(path.t);

    image.t = path.t;

    image.k = path.k;

   }

 else
  {
    radius = (1.0 / path.k);

    origin.x0 = path.x - radius * (sin(path.t));
```

```
        origin.y0 = path.y + radius * (cos(path.t));
        gamma = atan2(vehicle.y - origin.y0,
        vehicle.x - origin.x0);
        image.x = origin.x0 + fabs(radius) * (cos(gamma));
        image.y = origin.y0 + fabs(radius) * (sin(gamma));
        image.t = norm(gamma  + (PI/2)*(path.k/ fabs(path.k)));
        image.k = path.k;
    }
  return image;
  }
```

```
/******************************************************************
FUNCTION:  update_delta_d()
PARAMETERS: config, path
PURPOSE:   calculates the ystar for commanded_kappa()
RETURNS:   double
CALLED BY: commanded_kappa()
CALLS:     sin, cos
*******************************************************************/
double update_delta_d()
{

 double delta_d;

 CONFIGURATION path;

 path = current_robot_path.pc;

 delta_d = (-(vehicle.x - path.x) * (path.k *  (vehicle.x - path.x) + 2 * sin(path.t))
              -(vehicle.y - path.y) * (path.k *(vehicle.y - path.y) - 2 * cos(path.t)))
              / (1 + sqrt((path.k *(vehicle.x - path.x)+ sin(path.t))
              *(path.k *(vehicle.x - path.x)+sin(path.t))
             + ((path.k * (vehicle.y - path.y) - cos(path.t))
             *(path.k * (vehicle.y - path.y) - cos(path.t)))));

return delta_d;

} /* end update_delta_d() */
```

```
/******************************************************************
FUNCTION:  read_left_wheel_encoder
PARAMETERS: none
PURPOSE: Determines the distance moved by the left
            wheel by reading the optical encoder.
RETURNS: dist_l ( The distance moved by the left wheel
            in the current vehicle control cycle ).
CALLED BY:  evaluate_incremental_motion()
CALLS:      none
TASK: Level 4 interrupt
*******************************************************************/
double read_left_wheel_encoder()
{

double dist_l;

if (mv_direction > 0.0)
  dist_l = mv_direction * dlenc * ENC2DIST;

else

  dist_l = mv_direction * drenc * ENC2DIST;

return dist_l;

}/* end read_left_wheel_encoder */
```

```
/*************************************************************************
FUNCTION:  read_right_wheel_encoder
PARAMETERS: none
PURPOSE:  Determines the distance moved by the right
          wheel by reading the optical encoder.
RETURNS:  dist_r ( The distance moved by the right wheel
          in the current vehicle control cycle ).
CALLED BY:  evaluate_incremental_motion()
CALLS:     none
TASK: Level 4 interrupt
*************************************************************************/
double read_right_wheel_encoder()
{

double dist_r;

if (mv_direction > 0.0)
   dist_r = mv_direction * drenc * ENC2DIST;

else

   dist_r = mv_direction * dlenc * ENC2DIST;

return dist_r;

} /* read_right_wheel_encoder */
```

```
/*********************************************************************
FUNCTION:  pwm_lookup
PARAMETERS: vel (wheel velocity)
PURPOSE: Determines the estimated pwm ratio given the desired wheel velocity
            as an input.(This table get from 7.9 KHZ motor output curve)
RETURNS: pwm value based upon empirically determined velocity
            vs pwm ratio curve.
CALLED BY:  control()
CALLS:      none
TASK: Level 4 interrupt
*********************************************************************/
double pwm_lookup(vel)
double vel;
{

double v;

double pwm_value;

v = vel;

if (v == 0.0 )
pwm_value = 0.0;

else if (v >= 0.0 && v < 25.0)
pwm_value = (0.96 * v + 49.0);

else if (v >= 25.0 && v < 53.0)
pwm_value = (0.82 * (v - 25.0) + 73.0);

else if (v >= 53.0 && v <= 65.0)
pwm_value = (2.0 * (v - 53.0) + 96.0);

else if (v > 65.0)
pwm_value = 127.0;

else if (v < 0.0 && v >= - 2.5)
pwm_value = (1.2 * ( v ) - 54.0);

else if (v < -2.5 && v >= -13.0)
pwm_value = (0.76 * (v + 2.5) - 57.0);
else if (v < -13.0 && v >= -20.0)
pwm_value = (0.43 * (v + 13.0) - 65.0);

else if (v < -20.0 && v >= -34.0)
pwm_value = (1.0 * (v + 20.0) - 68.0);

else if (v < -34.0 && v >= -41.0)
pwm_value = (0.7 * (v + 34.0) - 82.0);

else if (v < -41.0 && v >= -49.0)
pwm_value = (1.5 * (v + 41.0) - 87.0);

else if (v < -49.0 && v >= -62.0)
pwm_value = (1.1 * (v + 49.0) - 99.0);

else if (v < -62.0 && v >= -65.0)
pwm_value = (2.3 * (v + 62.0) - 113.0);

else
pwm_value = -127.0;

return pwm_value;

} /* end pwm_lookup */
```

C. SPEED TEST PROGRAM

```
#include "mml.h"

/*******************************************************************
*This is a program of testing robot motor(forward)
*use this program to replace control.c of MML
* and use user.c which is provided at the end of this section
*to download the robot actual speed, this program will let robot go forward
*******************************************************************/
control_motor_test_forward()
{
 register int lpwm, rpwm;

 register double vl, vr, vel;

 register double delta_s, delta_theta;

 void store_loc_trace_data();

 double read_right_wheel_encoder();

 double read_left_wheel_encoder();

 double pwm_lookup();

 delta_s = (read_right_wheel_encoder() + read_left_wheel_encoder()) / 2.0;

 vel = delta_s / INTVL;

 if (pwm <  127.0)
 {

  pwm = pwm + 0.05;

 }
 else
 {

  pwm = 127.0;

 }

 rpwm = lpwm = (int)pwm;
 mcw = (mcw & 0xf0f0) | (lpwm > 0?1:2) | (rpwm > 0 ? 0x0100 : 0x0200);

 if (lpwm > 127)
      lpwm = 127;

 else if (lpwm < -127)
      lpwm = -127;

 if (rpwm > 127)
    rpwm = 127;

 else if (rpwm < -127)
      rpwm = -127;

 if (ltrace_f != 0)      /* trace loc  */
    store_loc_trace_data(pwm, vel);

 return (lpwm << 16 | rpwm & 0xff);

}/* end control_motor_test_increase */
```

```
/*******************************************************************
*This is a program of testing robot motor(backward)
*use this program to replace control.c of MML
* and use user.c which is provided at the end of this section
*to download the robot actual speed, this program will let robot go backward
*******************************************************************/
control_motor_test_backward()
{
 register int lpwm, rpwm;

 register double vl, vr, vel;

 register double delta_s, delta_theta;

 void store_loc_trace_data();

 double read_right_wheel_encoder();

 double read_left_wheel_encoder();

 double pwm_lookup();

 delta_s = (read_right_wheel_encoder() + read_left_wheel_encoder()) / 2.0;

 vel = delta_s / INTVL;

 if (pwm > - 127.0)
 {

  pwm = pwm - 0.05;

 }
 else
{

  pwm = -127.0;

 }

 rpwm = lpwm = (int)pwm;

 mcw = (mcw & 0xf0f0) | (lpwm > 0?1:2) | (rpwm > 0 ? 0x0100 : 0x0200);
```

```
    if (lpwm > 127)
        lpwm = 127;

    else if (lpwm < -127)
        lpwm = -127;

    if (rpwm > 127)
      rpwm = 127;

    else if (rpwm < -127)
        rpwm = -127;

    if (ltrace_f != 0)      /* trace loc  */
      store_loc_trace_data(pwm, vel);

    return (lpwm << 16 | rpwm & 0xff);

  }/* end control_motor_test_increase */
```

```
/******************************************************************
*This is a program of testing robot decreasing speed
*use this program to replace control.c of MML
* and use user.c which is provided at the end of this section
*to download the robot actual speed.
*the robot will go to 65 cm/s then decrease the speed to 0
******************************************************************/
control_motor_test_decrease()
{

 register int lpwm, rpwm;

 register double vl, vr, vel;

 register double delta_s, delta_theta;

 void store_loc_trace_data();

 double read_right_wheel_encoder();

 double read_left_wheel_encoder();

 double pwm_lookup();

 delta_s = (read_right_wheel_encoder() + read_left_wheel_encoder()) / 2.0;

 vel = delta_s / INTVL;

 if (vel >= 65.0)
   {

     top_speed = 1.0;

   }

 if (top_speed == 0.0)
   {

     if (pwm <  127.0)
      {

        pwm = pwm + 1.0;
```

```
    }
    else
    {

     pwm = 127.0;

    }
  }
  else
 {
   if (pwm > 0.0)
    {
     pwm = pwm - 0.05;
    }
   else
   {
     pwm = 0.0;
    }
}

rpwm = lpwm = (int)pwm;

mcw = (mcw & 0xf0f0) | (lpwm > 0?1:2) | (rpwm > 0 ? 0x0100 : 0x0200);

if (lpwm > 127)
        lpwm = 127;

else if (lpwm < -127)
        lpwm = -127;

if (rpwm > 127)
     rpwm = 127;

else if (rpwm < -127)
        rpwm = -127;

if (ltrace_f != 0)      /* trace loc  */
     store_loc_trace_data(pwm, vel);

return (lpwm << 16 | rpwm & 0xff);

}/* end control_motor_test_decrease */
```

```
/*********************************************************************
*This is a program of testing robot speed(forward)
*use this program to replace control.c of MML
* and use user.c which is provided at the end fo this section
*to download the curve of  robot actual speed v.s. desired speed
*********************************************************************/
control_velocity_test_forward()
{

 register int lpwm, rpwm;

 register double vl, vr, vel;

 register double delta_s, delta_theta;

 void store_loc_trace_data();

 double read_right_wheel_encoder();

 double read_left_wheel_encoder();

 double pwm_lookup();

 delta_s = (read_right_wheel_encoder() + read_left_wheel_encoder()) / 2.0;

 vel = delta_s / INTVL;

 rpwm = lpwm = pwm_lookup(desired_vel) + kpw_b * (desired_vel - vel);

 mcw = (mcw & 0xf0f0) | (lpwm > 0?1:2) | (rpwm > 0 ? 0x0100 : 0x0200);

 if (lpwm > 127)
      lpwm = 127;

 else if (lpwm < -127)
      lpwm = -127;

 if (rpwm > 127)
    rpwm = 127;

 else if (rpwm < -127)
      rpwm = -127;
```

```
  if (desired_vel <= -65.0)
   {

     desired_vel = -65;

   }
  else
   {

     desired_vel = desired_vel - 0.01;

   }

 if (ltrace_f != 0)      /* trace loc  */
     store_loc_trace_data( vel, vel, vel, vel);

 return (lpwm << 16 | rpwm & 0xff);

}/* end control_velocity_test */
```

```
/*********************************************************************
*This is a program of testing robot speed(backward)
*use this program to replace control.c of MML
* and use user.c which is provided at the end fo this section
*to download the curve of  robot actual speed v.s. desired speed
*********************************************************************/
control_velocity_test_backward()
{

 register int lpwm, rpwm;

 register double vl, vr, vel;

 register double delta_s, delta_theta;

 void store_loc_trace_data();

 double read_right_wheel_encoder();

 double read_left_wheel_encoder();

 double pwm_lookup();

 delta_s = (read_right_wheel_encoder() + read_left_wheel_encoder()) / 2.0;

 vel = delta_s / INTVL;

 rpwm = lpwm = pwm_lookup(desired_vel) + kpw_b * (desired_vel - vel);

 mcw = (mcw & 0xf0f0) | (lpwm > 0?1:2) | (rpwm > 0 ? 0x0100 : 0x0200);

 if (lpwm > 127)
      lpwm = 127;

 else if (lpwm < -127)
      lpwm = -127;

 if (rpwm > 127)
    rpwm = 127;

 else if (rpwm < -127)
      rpwm = -127;
 if (desired_vel >= 65.0)
   {

     desired_vel = 65;

   }
 else
   {

     desired_vel = desired_vel + 0.01;

   }

 if (ltrace_f != 0)      /* trace loc  */
     store_loc_trace_data( vel, vel, vel, vel);

 return (lpwm << 16 | rpwm & 0xff);

}/* end control_velocity_test */
```

```
/******************************************************************
*This is a user.c program for testing robot speed
*
******************************************************************/

#include "mml.h"
user_speed_test()
{
 CONFIGURATION start, second, third, circle;

 buffer_loc = index_loc = malloc(300000);

 bufloc = indxloc = (double *)malloc(60000);

 loc_tron(2,0x3f,30);

 def_configuration(0.0, 0.0, 0.0, 0.0, &start);

 set_rob(&start);

 line(&start);

 wait_timer(6500);

 loc_troff();

 halt();

 motor_on = 0;

 loc_trdump("actual-speed-vs-desired-speed");

}
```

D. USER PROGRAM O[?] FINAL RESULTS

```
/*********************************************************************
*This is a user.c program for line to circle to line motion
*
**********************************************************************/
#include "mml.h"

user()
{
 CONFIGURATION start, second,circle;

 buffer_loc = index_loc = malloc(300000);

 bufloc = indxloc = (double *)malloc(60000);

 loc_tron(2,0x3f,5);

 def_configuration(0.0,-10.0, 0.0, 0.0, &start);

 def_configuration(0.0, 10.0, PI, 0.0, &second);

 def_configuration(200.0, -50.0, 0.0, 0.02, &circle);

 set_rob(&start);

 speed(20.0);

 line(&start);

 line(&circle);

 line(&second);

 while (vehicle.x >= 0.0);

 stop0();

 loc_troff();

 loc_trdump("line-circle-line-tracking.05Feb94");

}
```

```
/************************************************************************
*This is a user.c program for a star-shaped motion
*
************************************************************************/
#include "mml.h"

user()
{
 CONFIGURATION start,one,two,three,four,five;

 double angle1;

 double DISTANCE;

 double DIST1;

 double DIST2;

 DISTANCE = 110.0;

 DIST1 = 150.0;

 DIST2 = DIST1 - DISTANCE;

 r_printf(" \12 This is a star-motion program.");


 def_configuration(0.0,0.0,0.0,0.0, &start);

 def_configuration(DISTANCE,0.0,0.0,0.0, &one);

 def_configuration(DIST1-cos(PI/5)*DISTANCE,sin(PI/5)*DISTANCE,
                                                      2.51,0.0, &two);
 def_configuration(DIST1-cos(PI/5)*DIST1 + sin(PI/10)*DISTANCE,
                   sin(PI/5)*DIST1-cos(PI/10)*DISTANCE,5.02,0.0, &three);
 def_configuration(cos(PI/5)*DIST1-sin(PI/10)*DIST2,
                   sin(PI/5)*DIST1-cos(PI/10)*DIST2,7.53,0.0, &four);

 def_configuration(cos(PI/5)*DIST2,sin(PI/5)*DIST2,10.04,0.0, &five);

 set_rob(&start);
speed(10.0);

buffer_loc = index_loc = malloc(300000) ;

bufloc = indxloc = (double *) malloc(60000);

loc_tron(2,0x3f,5);

bline(&one);

bline(&two);

bline(&three);

bline(&four);

line(&five);

while(vehicle.x>=0.0);

stop0();

loc_troff();

loc_trdump("start-motion.05Feb94");

}
```

```
/*******************************************************************
*This is a user.c program for path tracking mission
*
*******************************************************************/

#include "mml.h"

user()
{
 CONFIGURATION start;

 CONFIGURATION first;

 CONFIGURATION second;

 CONFIGURATION third;

 def_configuration(0.0, 0.0, 0.0, 0.0, &start);

 def_configuration(0.0, 50.0,0.0, 0.0, &first);

 def_configuration(200.0, 150.0, HPI, 0.0 ,&second);

 def_configuration(250.0, 350.0, HPI, 0.0, &third);

 buffer_loc = index_loc = malloc(300000);

 bufloc = indxloc = (double *)malloc(60000);

 loc_tron(2,0x3f,10);

 set_rob(&start);

 speed(20.0);

 line(&first);

 line(&second);

 while (vehicle.y <150);

 skip();
```

```
  bline(&third);

  while (vehicle.y <350);

  stop0();

  loc_troff();

  loc_trdump("Path-tracking.27Jan94");

}
```

```
/*********************************************************************
*This is a user.c program for obstacle avoidance
*
*********************************************************************/

#include "mml.h"

user()
{
 double hit11;

 double hit12;

 CONFIGURATION  p1, p3,start;

 def_configuration(0.0, 0.0, 0.0, 0.0, &start);

 def_configuration(500.0, 0.0, 0.0, 0.0, &p1);

 def_configuration(500.0,-100.0,0.0,0.0, &p3);

 buffer_loc = index_loc = malloc(300000) ;

 bufloc = indxloc = (double *) malloc(60000);

 loc_tron(2, 0x3f, 30);

 hit11 = 9999.9;

 hit12 = 9999.9;

 set_rob(&start);

 speed(20.0);

 enable_sonar(FRONTR);

 hit11 = sonar(FRONTR);

 line(&p1);
```

```
while(hit11 >= 100.0 || hit11 == 0.0 )
{
  hit11 = sonar(FRONTR);
}

skip();

line(&p3);

enable_sonar(LEFTF);

hit12 = sonar(LEFTF);

while(hit12 >= 999.0 || hit12 == 0.0 )
{
  hit12 = sonar(LEFTF);
}

while(hit12 <= 150.0)
{
  hit12 = sonar(LEFTF);
}

while(hit12 >= 999.0 || hit12 == 0.0 )
{
  hit12 = sonar(LEFTF);
}

while(hit12 <= 150.0)
{
  hit12 = sonar(LEFTF);
}

skip();

bline(&p1);

while (vehicle.x < 500.0);

disable_sonar(FRONTR);

disable_sonar(LEFTF);
```

```
    loc_troff();

    motor_on = 0;

    loc_trdump("obstacle-avoidance.07Feb94");

   }
```

LIST OF REFERENCES

1. Richard James Abresch, "Path Tracking Using Simple Planar Curves", Naval Postgraduate School, March, 1992.

BIBLIOGRAPHY

2. Yutaka Kanayama, "Mathematical Theory of Robotics: Introduction to 2D Spatial Reasoning", Naval Postgraduate School, December, 1993.

3. Yutaka Kanayama, Masanori Onishi, "Locomotion Functions in the Mobile Robot Language, MML", IEEE International Conference on Robotics and Automation, April, 1991.

4. James Alan Alexander, "Motion Control and Obstacle Avoidance for Autonomous Vehicles Using simple Planar Curves", Naval Postgraduate School, March, 1993.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Dudley Knox Library 2
   Code 052
   Naval Postgraduate School
   Monterey, CA 93943-5002

3. Chairman, Code CS 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

4. Dr Yutaka Kanayama, Code CS/KA 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

5. Dr Yuh-jeng Lee, Code CS/KA 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

6. LTCOL Ten-Min Lee 2
   #103, 225 LN, Chung-Shing RD,
   Nantou, Taiwan, R.O.C.